UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Ingeniería de Telecomunicación

PROYECTO FIN DE CARRERA

Desarrollo de un sistema Web 2.0 para gestión de contenidos musicales con integración de datos de múltiples fuentes externas

Autor: Francisco Javier Guzmán Rivas
Tutor: Jesús Arias Fisteus

ACCESO CÓDIGO FUENTE

MADRID, 6 DE NOVIEMBRE DE 2008

PROYECTO FIN DE CARRERA

Departamento de Ingeniería Telemática

Universidad Carlos III de Madrid

Título: Desarrollo de un sistema Web 2.0 para gestión de contenidos musicales con integración de datos de múltiples fuentes externas
Autor: D. Francisco Javier Guzmán Rivas
Tutor: D. Jesús Arias Fisteus

La defensa del presente proyecto fin de carrera se realizó el día 6 de noviembre de 2008 siendo calificada por el siguiente tribunal:

Presidente:

Vocal:

Secretario:

Habiendo obtenido la siguiente calificación

Presidente **Vocal** **Secretario**

Agradecer a mi familia su apoyo y comprensión en la decisión que me llevó a continuar mis estudios.
Valorar la comprensión, cariño, apoyo y aguante de Vanessa.
Aguantar mis charlas interminables sobre el proyecto, pero sobretodo por escucharlas y ofrecer su punto de vista.
Y claro por estar a mi lado, así todo es más sencillo.
Agradezco a mi amigo/compañero Alberto su colaboración.
Recordar a toda la gente que coincidió conmigo en la universidad.
No me olvido de agredecer a mi tutor Jesús sus opiniones y valoraciones, y también la libertad que me ha dado a lo largo de este año de trabajo.

Resumen

En los útlimos años ha surgido un nuevo tipo de aplicaciones web que se han denominado Aplicaciones Web 2.0. En este tipo de aplicaciones los usuarios tienen una participación muy activa en comparación con las anteriores aplicaciones en las que los usuarios eran meros espectadores.

Actualemente la cantidad de datos y aplicaciones que utilizan los usuarios es mucho mayor debido al desarrollo de la tecnología y la evolución hacia conexiones de banda ancha. Por lo tanto, surge la necesidad de la gestión e integración de los datos del usuario y de los datos de otras aplicaciones en una única aplicación.

Este proyecto surge con el objetivo de estudiar varias de las herramientas que permiten reducir la complejidad del desarrollo de aplicaciones de la Web 2.0. En concreto, se presenta el caso de estudio de la creación de una aplicación que permita la gestión de la información musical que proporcionen los usuarios. Esta información estará disponible desde cualquier lugar con conexión a Internet. Además, la aplicación complementará la información proporcionada por los usuarios con información (letras, vídeos, productos relacionados, etc.) obtenida de otras fuentes, constituyendo lo que se denomina un *mashup*.

Este proyecto está constituido por dos aplicaciones: una aplicación de escritorio y una aplicación web. La aplicación de escritorio envía la información del usuario a la aplicación web, y la mantiene sincronizada. La aplicación web permite la consulta y gestión de dicha información, enriquecida con información complementaría proveniente de otras fuentes/aplicaciones.

La aplicación web ha sido desarrollada con el *framework* Ruby on Rails. Con este *framework* se ha desarrollado una aplicación orientada a REST que facilita la gestión de los recursos de información de los usuarios y la comunicación con la aplicación de escritorio. La aplicación de escritorio se ha desarrollado en Java, y permite tanto procesar la información musical de usuarios del reproductor iTunes y Amarok como obtenerla directamente desde directorios que contengan ficheros de música.

Abstract

In the last few years, a new kind of Web applications called Web 2.0 Applications has emerged. In these applications users have a very active participation in comparison with previous applications in which users were mere spectators.

Nowadays people can use a much higher amount of data and applications thanks to the development of technology and the evolution towards broadband connections. Therefore, the need for the management and integration of user's data and data from other applications into a single application arises.

This project comes up with the aim of studying several of the existing tools that make the reduction of the complexity of Web 2.0 application development possible. Specifically, it presents a case study consisting in the development of an application that enables the management of musical information provided by users. This information will be available from anywhere through an Internet connection. In addition, the application will complement the information provided by users with data obtained from other sources or applications such as lyrics, videos or related products, giving as a result what it is called a mashup.

The project consists of a desktop application and a web application. The desktop application sends user information to the web application, and keeps it in sync. The Web application allows web consulting and management of such information, enriched with data from other sources/applications.

The web application has been developed using the Ruby on Rails framework. With this framework a REST-oriented application has been obtained, making easier the related resource management feedback and the communication with the desktop application. The desktop application has been developed in Java, and allows not only the processing of information from users of music players iTunes and Amarok but also getting it directly from directories that contain music files.

Índice

Índice de Figuras

Abreviaturas y Acrónimos

AJAX Asynchronous JavaScript And XML. 8, 11

API interfaz de programación de aplicación. 2, 5–8, 10, 11, 22, 28, 29, 44, 50, 55, 57, 64, 69, 75, 77, 78, 81–83, 85, 97

AWS Amazon Web Services. 57

CoC Convention over Configuration. 14, 15

CRUD create, read, update and delete. 9, 19, 20

DRY Don't Repeat Yourself. 9, 14, 15

DTD Document Type Definition. 63

HTML HyperText Markup Language. 6, 9, 17, 24, 26

HTTP Hypertext Transfer Protocol. 8–10, 50, 60, 67, 73, 76, 91

ID3 Específicación de un contenedor de métadatos. 65, 75

JAXB Java API for XML Binding. 77, 79

JS JavaScript. 24

JSON JavaScript Object Notation. 18, 24, 66, 97

JSP Java Server Pages. 24

LGPL GNU Lesser General Public License. 65

MIME Multipurpose Internet Mail Extensions. 91

MP3 MPEG-1 Audio Layer 3. 64, 65, 75

MVC Modelo Vista Controlador. 11, 14–16, 36, 71, 95

PDA personal digital assistant. 65

PHP PHP Hypertext Pre-processor. 14

REST REpresentational State Transfer. 2, 5, 8–10, 24, 29, 35, 36, 38, 39, 42, 44, 50, 55, 62, 63, 66, 67, 69–71, 81, 82

RJS Ruby JavaScript. 24, 88, 89

RSS Really Simple Syndication. 9, 84, 85

SOAP Simple Object Access Protocol. 5, 10, 38, 55, 82, 83

SQL Structured Query Language. 16, 21

URI Uniform Resource Identifier. 51

URL Uniform Resource Locator. 9, 10, 17, 38, 42, 50–56, 66, 72, 74, 81, 87

W3C World Wide Web Consortium. 10

WSDL Web Service Description Language. 57, 83

XML Extensible Markup Language. 9, 18, 24, 50, 63, 64, 66, 68, 77–79

CAPÍTULO 1

INTRODUCCIÓN

1.1. Motivación

En los últimos años el incremento de la utilización de Internet y la simplificación de las tecnologías de desarrollo de aplicaciones ha dado lugar a la aparición de multitud de aplicaciones web de muy diversa índole. Y una de las temáticas frecuentes en las aplicaciones web son aquellas que contienen información musical.

También es necesario destacar la evolución que han sufrido las aplicaciones web desde unas aplicaciones que constituían meros escaparates de productos e información estática (*Web 1.0*) hasta aplicaciones web cuya información es generada o proporcionada por los propios usuarios de las aplicaciones (*Web 2.0*). Los usuarios ya no son meros clientes de las aplicaciones sino que ahora son clientes y servidores de información.

Muchas de las aplicaciones de la *Web 2.0* se utilizan como almacén de información por parte de sus usuarios. A partir de la información almacenada por sus usuarios estas aplicaciones aportan nuevos y muy diferentes servicios. De entre la información que es posible almacenar se encuentran: páginas favoritas, fotos, información de contactos, información musical, etcétera.

Con este proyecto se proporciona a los usuarios la capacidad de almacenar los datos de su biblioteca de información musical para que puedan acceder a ella desde un navegador con conexión a Internet.

Otro aspecto a destacar de muchas aplicaciones actuales es que disponen de interfaces públicas que permiten el acceso a la información que alojan. Entre estas aplicaciones se encuentran: Google, Yahoo, YouTube, Amazon, Lastfm, etcétera. Esto va a permitir que otras aplicaciones generen nuevos servicios con un valor añadido a partir de la información pública disponible a través de dichas interfaces. Las aplicaciones que se construyen siguiendo este esquema se denominan aplicaciones híbridas o *mashups*[1]. Estas aplicaciones tienen también éxito como fue el caso de Panoramio, la cual fue adquirida por Google.

En este proyecto se obtendrá la información alojada en diferentes aplicaciones web para complementar la información que almacenan los usuarios en la aplicación web acerca de su biblioteca musical. De este modo la aplicación no solo se comportará como un almacén de información sino que también proporcionará información relevante para los usuarios de la misma.

Otra de las características de éxito en las aplicaciones que forman la denominada *web 2.0* es la de proporcionar capacidad de interacción entre los usuarios de dichas aplicaciones. Esta característica da lugar a la formación de redes sociales. Las redes sociales pueden ser redes sociales temáticas o no. En el caso de este proyecto la red social se formará entorno a información musical.

[1] A lo largo del texto se utilizará la voz inglesa *mashup* dada su gran aceptación.

1.2. Objetivos

El objetivo inicial de este proyecto fin de carrera es la exploración de diferentes *frameworks* de desarrollo web enfocados a la creación de *mashups* y la integración de datos procedentes de fuentes heterógeneas. Del *framework* elegido se explorarán sus características y funcionalidades utilizándolo de forma aplicada a un caso de estudio. Este caso de estudio está dirigido a la creación de un *mashup* de gestión información musical.

A continuación se enumeran los objetivos que cubre este caso de estudio:

- Desarrollar un sitio Web que permita a los usuarios subir y navegar los datos de sus colecciones musicales.
- Desarrollar un sitio Web con una interfaz pública accesible por otras aplicaciones.
- Relacionar las colecciones del usuario con otras fuentes de información disponibles en la Web (caratulas, letras, vídeos, productos de tiendas en línea, descargas, etcétera.)
- Utilizar y/o combinar el contenido proveniente de las diferentes fuentes de información para presentarlo de un nuevo modo o para crear nueva información relevante para los usuarios.
- Proporcionar funciones de red social entre los usuarios.

1.3. Organización de la memoria

El presente documento se divide en los capítulos que se enumeran a continuación:

Introducción. Se realiza una introducción a las aplicaciones web de contenido musical junto con la motivación que lleva a la realización de este proyecto. Se enumeran los objetivos y organización del mismo.

Estado del Arte. En este apartado se expone la situación actual de las aplicaciones web. La *web 2.0* y las tecnologías que utiliza, la consulta de APIs de terceros y el desarrollo de aplicaciones con orientación REST (REpresentational State Transfer). La metodología de desarrollo ágil y algunos de los *frameworks* de desarrollo web que se utilizan para seguir dicha metodología: Grail y Ruby On Rails.

Diseño. En este capítulo se expone en primer lugar una visión del diseño global. Seguidamente se exponen las decisiones de diseño tomadas para cada una de las partes principales que componen todo el sistema: información proporcionada por los usuarios, aplicación web y aplicación de escritorio.

Implementación. En este capítulo se exponen los aspectos más relevantes de la implementación del trabajo como son: información proporcionada por los usuarios, el sistema de gestión de usuarios, aplicación de escritorio y aplicación web.

Validación y Pruebas. En este capítulo se enumeran de forma breve las pruebas automáticas y de validación realizadas sobre el desarrollo.

Conclusiones. En este capítulo se exponen las conclusiones alcanzas en este trabajo.

Trabajos Futuros. En este capítulo se enumeran los posible trabajos futuros que pueden ampliar o mejorar las funcionalidades implementadas.

Anexos. En este capítulo se encuentra un anexo relativo al manual de instalación de un entorno de desarrollo como el utilizado para elaborar este proyecto y los pasos necesarios para la puesta en marcha de la aplicación en dicho entorno de desarrollo. También se incluye un manual de uso de la aplicación con imágenes que muestran la utilización de la misma. Por último, se incluye un anexo con ejemplos de aspecto relevantes y código fuente de algunos ficheros que conforman el proyecto.

CAPÍTULO 2

ESTADO DEL ARTE

COMO ya se ha comentado en los puntos anteriores, este proyecto consiste en el desarrollo de una aplicación web que permita gestionar contenido musical que proporcionen los usuarios de la misma y complementar dicha información gracias al contenido proporcionado por otras aplicaciones web existentes.

Se abordará el diseño en primer lugar, y la implementación en segundo lugar, de un *mashup* de gestión de contenido musical.

En este capítulo se introduce el estado actual del desarrollo de este tipo de aplicaciones, explicando en primer lugar lo que se ha denominado **Web 2.0** y las redes sociales con sus tecnologías asociadas. Se comentará el desarrollo de aplicaciones REST en detrimento de la utilización de servicios web mediante SOAP (Simple Object Access Protocol). También se estudiarán los dos diferentes estilos de creación de *mashups* que se utilizan contemporáneamente a la elaboración de este proyecto

Posteriormente se aborda el estado del arte en el ámbito de los *frameworks* de desarrollo web. Se comentará en qué consisten estos *frameworks* y qué aportan a la hora de desarrollar aplicaciones web, y también se explicarán en mayor detalle los dos *frameworks* que se valoraron para la implementación de este proyecto. Dentro de este mismo punto se detalla una tendencia de desarrollo web muy utilizada para este y otros tipos de aplicaciones que se denomina *«Desarrollo Web Ágil»*.

Seguidamente se introducen las diferentes interfaces de programación de aplicaciones (APIs) que proporcionan información de contenido musical a través de diferentes servicios como pueden ser: obtener la letra de una canción, obtener artistas similares a uno dado, obtener la localización de álbumes de un artista en tiendas de música *online*, etcétera.

Por último, se comentaran algunas de las aplicaciones, *mashups* y redes sociales que constituyen el estado del arte de la aplicaciones web de contenido musical presentes en la web en el momento de la realización de este proyecto. Algunas de estas aplicaciones desaparecieron a lo largo de la realización del proyecto, siendo absorbidas por algunas de las restantes: Sin embargo, todas las que se comentan en este documento han sido valoradas.

2.1. Web 2.0 - Tecnologías

La idea original de la web (lo que ahora se denominará Web 1.0) se corresponde con un conjunto de páginas HTML (HyperText Markup Language) enlazadas entre sí y que no eran actualizadas frecuentemente.

El objetivo fundamental de la Web 1.0 era ser el escaparate de las empresas, buscando conseguir que a sus páginas accedieran el mayor número posible de usuarios para venderles sus productos. Eran meros escaparates publicitarios, páginas estáticas que no permitían ningún tipo de retroalimentación por parte de los usuarios que las visitaban. Estas empresas pensaban que la web era un canal más para su publicidad como podía ser la televisión, un canal unidireccional.

El término Web 2.0 fue acuñado por Dale Dougherty (de O'Reilly Media) en el año 2003. Este término no hace referencia a una nueva versión de la web, ni a nuevos protocolos, ni a nuevos lenguajes de programación, sólo es una nueva forma de hacer las cosas. Las aplicaciones y los servicios de la Web 2.0 se sustentan en estándares que existían antes de que se acuñara este término.

Tres principios recogen la definición del término Web 2.0:

- Tecnología: la evolución de la tecnología ha permitido que los usuarios tengan un papel activo en la generación del contenido de los distintos sitios web.
- Arquitectura modular: se favorece de este modo la creación de aplicaciones más complejas con un menor coste y de forma más rápida. Esta modularidad queda de manifiesto en las APIs que exponen diferentes sitios web.
- Comunidad: el usuario aporta contenidos, crea relaciones con otros usuarios, crea redes de conocimiento, se fomentan las redes sociales.

Con la Web 2.0 los usuarios aportan valor a los sitios web a través de la creación de contenido y de las relaciones que se establecen entre los usuarios. Se produce una descentralización de Internet, ahora cada cliente es también un servidor, entendido como creador de contenido.

En comparación con la Web 1.0 en la que los usuarios eran meros destinatarios de los contenidos, en la Web 2.0 los usuarios participan. Participan en el desarrollo de *software* (por ejemplo, en el desarrollo de Linux participan miles de personas, en contra de la forma de desarrollo por ejemplo de Windows Vista[1]). Los usuarios producen los contenidos, es caso de los *blogs* o las *wikis*. Los usuarios también participan de los negocios (Amazon).

A continuación se muestra una tabla comparativa de la Web 1.0 y la Web 2.0[2].

Web 1.0	**Web 2.0**
Páginas personales	Bitácoras
Especulación con nombres de dominio	Optimización de buscadores
Páginas vistas	Coste por clic
Informar	Participar, compartir
Sistemas de gestión de contenidos	Wikis
Directorios (taxonomía)	Etiquetas (folksonomía)
Fidelización	Sindicación
Publicidad con *banners* y *pop-ups*	Publicidad contextual

[1] Nombre registrado
[2] Fuente [1]

2.1.1. Consulta de APIs de terceros

Algunas de las aplicaciones de la Web 2.0 están compuestas por pequeñas piezas enlazadas. Son aplicaciones que utilizan y/o combinan los datos procedentes de una o más fuentes de información para presentarlos de una nueva manera. Los dueños de los datos facilitan el acceso a los mismos por parte de terceros. Este acceso lo posibilitan las APIs. A este tipo de aplicaciones se les denomina híbridas (o *mashups*)[3].

En este capítulo no se entra en el detalle de cómo se accede a esta información, pero se presentan dos de las aproximaciones o estilos más comunes para el desarrollo de *mashups*.

Los principales estilos de creación de un *mashup* son: *mashups* de lado de servidor (*server-side mashups*) y *mashups* de lado del cliente (*client-side mashups*)[4].

En *mashups* de lado de servidor se integran los servicios y contenidos en el servidor. El servidor actúa como *proxy* entre una aplicación web en el cliente, generalmente un navegador, y el otro sitio web del que se obtiene contenido para constituir el *mashup*. En este tipo de *mashups* las peticiones del cliente van contra el servidor de la aplicación, el cual actúa como *proxy* realizando las llamadas oportunas al sitio web (de un tercero) que aporta el contenido. En general, un *mashup* de este tipo trabaja según muestra la figura 2.1[5].

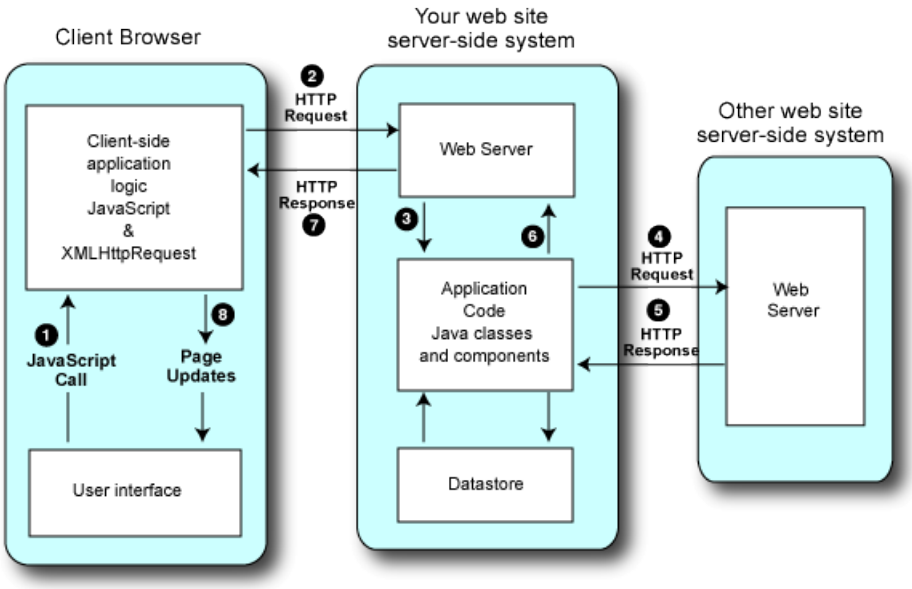

Figura 2.1: *Mashup* de lado de servidor

Por otro lado, en los *mashups* de lado de cliente se integran los servicios y el contenido en el mismo cliente. Se integra directamente el contenido o funcionalidad de los sitios web consultados. En este tipo de *mashups* el propio cliente puede realizar peticiones directamente al otro sitio web. En general, este tipo de *mashups* trabajan según muestra la figura 2.2[6].

En este proyecto se ha optado por realizar un *mashup* de lado de servidor. En primer lugar, porque una gran parte de los servicios a los que se accede cuentan con una biblioteca ya desarrollada que permite su utilización de forma sencilla.

[3] A lo largo del texto se utilizará la voz inglesa de *mashup* para referirse a este tipo de aplicaciones
[4] Para ampliar información consultar [2] las imágenes mostradas han sido obtenidas del mismo artículo de referencia
[5] La figura 2.1 se tomó del artículo [2].
[6] La figura 2.2 se tomó del artículo [2].

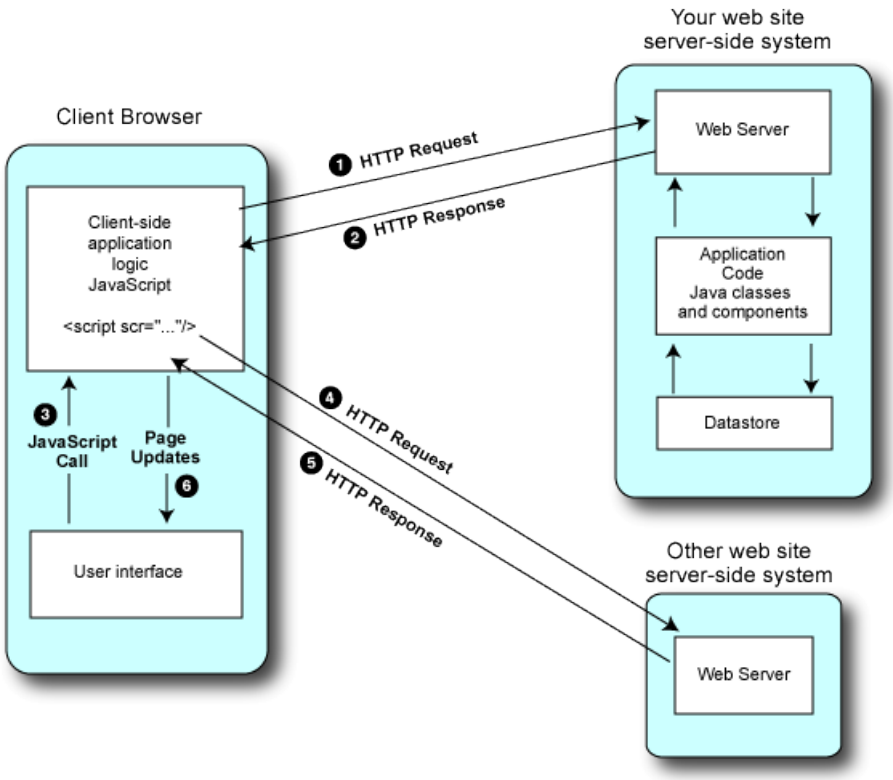

Figura 2.2: *Mashup* de lado de cliente

En segundo lugar, si se quiere que el acceso a este contenido sea dinámico, asíncrono, es decir, se quiere utilizar el objeto `XMLHttpRequest`, el cual es el núcleo de las aplicaciones web actuales basadas en AJAX (Asynchronous JavaScript And XML)[7], se debe saber que si se escribe código en cliente que utilice este objeto se encuentran varias restricciones impuestas por los navegadores web, entre ellas una que impide realizar conexiones a otros dominios (*cross-domain connections*). Para resolver este problema existen diferentes soluciones como se muestra en [3], pero esto no es necesario si se construye un *mashup* del lado de servidor como se ha decidido.

Con esto último, también se evita lo que se conocen como *«clientes pesados»*, es decir, se evita cargar gran parte de la lógica de la aplicación en las aplicaciones cliente, generalmente los navegadores web.

La existencia de los *mashups* radica fundamentalmente en la facilidad del acceso a sus datos que proporcionan terceros a través de sus APIs. El siguiente apartado se centra en la presentación de una tecnología que permite desarrollar aplicaciones o al menos la interfaz para que expongan sus datos de forma estructurada y que éstos sean accesibles a través de un API.

2.1.2. Desarrollo de aplicaciones REST

Desde un punto de vista práctico, cuando se habla de construir una aplicación orientada a REST, lo que se quiere decir es que se va a construir una interfaz simple hacia la aplicación donde con el uso de los cuatro métodos fundamentales de HTTP (HyperText Transfer Protocol) (GET, POST, PUT y DELETE) se definen las acciones que se pueden llevar a cabo sobre los recursos (información) que maneja la aplicación[8].

[7] Consultar Wikipedia AJAX - http://es.wikipedia.org/wiki/AJAX
[8] Imagen procedente de peepcode.com

```
GET    "/pages"          # => index
POST   "/pages"          # => create

GET    "/pages/FAQ"      # => show
PUT    "/pages/FAQ"      # => update
DELETE "/pages/FAQ"      # => destroy

GET    "/pages/new"      # => new
GET    "/pages/FAQ/edit" # => edit
```

Figura 2.3: Utilización métodos HTTP

En REST todo recurso es una entidad direccionable mediante una URL (Uniform Resource Locator) única con la que se puede interaccionar a través del protocolo HTTP. Los recursos pueden ser representados en distintos formatos como HTML, XML (Extensible Markup Language), RSS (Really Simple Syndication) según lo solicite el cliente que consulta la aplicación[9].

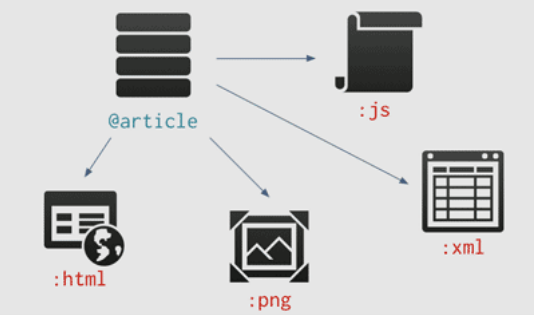

Figura 2.4: Un recurso múltiples representaciones

A continuación se muestra una lista de características de las aplicaciones REST.

- URLs limpias. En REST las URLs no representan acciones sino recursos. La manipulación requerida es independiente de la URL y se expresa con la ayuda de los métodos HTTP.
- Formatos de respuesta variados (Figura 2.4). Una misma acción de los controladores REST puede entregar HTML, XML, RSS o cualquier otro formato de datos según los requisitos de la aplicación cliente de manera simple y sencilla.
- Menos código. Al desarrollar acciones únicas para diferentes formatos evita repeticiones en el sentido DRY (Don't Repeat Yourself)[10], y como resultado los controladores tienen menos código.
- Controladores orientados a CRUD (Create Read Update and Delete). Los controladores y los recursos se funden en una única cosa. Cada controlador tiene como responsabilidad manipular un único tipo de recurso.
- Diseño limpio de la aplicación. El desarrollo REST produce un diseño de aplicación conceptualmente claro y más fácil de entender, gracias a las características anteriores.

[9] Imagen procedente de peepcode.com
[10] Será explicado más adelante en el texto

Hasta la fuerte entrada del paradigma REST para la creación del API de las diferentes aplicaciones la tecnología que era utilizada y que aún lo sigue siendo son los servicios web implementados mediante SOAP.

SOAP define un protocolo de intercambio de información estructurada y tipada entre pares en un entorno distribuido y descentralizado. El intercambio de información a través de la red es habitual que se realice sobre el protocolo de transporte HTTP, aunque no es necesaria su utilización estrictamente, pudiéndose utilizar otros protocolos para el transporte de información.

Dentro de las ventajas de SOAP[11] tenemos:

- Es independiente de la plataforma. Lo mismo ocurre con REST.
- Es independiente del lenguaje. Al igual que REST.
- Es sencillo y extensible. Punto discordante con REST.

Las dos primeras ventajas que presenta SOAP las comparte también el paradigma REST. Ambas tecnologías son independientes tanto de la plataforma como del lenguaje, si cabe SOAP es más independiente en ambos sentidos ya que está sujeta a una estandarización pero no es relevante en este caso.

En cuanto a la tercera ventaja aparecen las primeras diferencias sustanciales entre REST y SOAP. Hasta la llegada de REST, SOAP podía ser considerado un protocolo sencillo gracias fundamentalmente a que es un estándar del World Wide Web Consortium (W3C) y esto facilita enormemente su implementación y utilización. Sin embargo, REST es mucho más sencillo.

La anterior afirmación se sustenta en que SOAP al constituir un protocolo necesita estructurar la información tanto para la llamada a procedimientos remotos como para el intercambio de información. En la llamada a un procedimiento remoto mediante SOAP es necesario construir lo que se denomina un *envelope* (sobre), donde se alojará de forma estructurada toda la información necesaria para invocar un servicio remoto. Esto provoca un mayor uso del ancho de banda, y mayor consumo de recursos tanto en el cliente como en el servidor y es más complejo que la simple generación de una petición a una URL específica, como resultaría de la utilización de REST. Por lo tanto, la afirmación de que SOAP es un protocolo sencillo debe precisarse sobre qué escenario lo es.

Sin embargo, el otro aspecto fundamental que diferencia a SOAP de REST es la capacidad de extender o ampliar sus capacidades. Aquí REST entra en una fuerte desventaja frente a SOAP, de aquí que el uso de REST se enfoque a tareas más sencillas. SOAP permite definir nuevos protocolos para el envío y procesado de mensajes como puede ser: seguridad, cifrado, fiabilidad, etcétera.

Para ampliar información sobre la tecnología REST consultar las referencias [4], [5] y [6]. En el caso de SOAP consultar [7], [8] y [9].

2.2. Frameworks de desarrollo web

En los últimos años ha crecido considerablemente el desarrollo de aplicaciones web dada la gran expansión que ha tenido Internet en todo el mundo. A su vez, en los últimos dos o tres años se ha establecido una tendencia muy clara hacia la utilización de *frameworks* para el desarrollo de aplicaciones web. Con estos *frameworks* se pretende establecer esquemas de trabajo basados en patrones de programación y normas que, en la práctica, conducen al aprovechamiento del tiempo, optimización de las tareas y la perdurabilidad y escalabilidad de las aplicaciones.

Estos *frameworks* o moldes lo que intentan es evitar tareas reiteradas, dando soluciones a problemas que se dan en el desarrollo de toda aplicación web y minimizando las posibilidades de cometer errores.

[11] Información obtenida de http://en.wikipedia.org/wiki/SOAP

A continuación se exponen algunas de las características generales que resultan comunes a la mayoría de los *frameworks*.

1. **Convención.** Mediante una colección de convenciones, se permite que el *framework* resuelva decisiones que si no debería tomar el desarrollador, y que podrían llevar a la pérdida de tiempo e incluso a la introducción de errores. Un ejemplo bien claro se da en la elección de la arquitectura de la aplicación y la definición de sus capas. En la mayoría de los *frameworks* lo resuelven apostando por un patrón Modelo Vista Controlador (MVC).

2. **Modularidad.** Desde prácticamente el inicio del desarrollo de software se ha intentado separar las distintas partes de las aplicaciones en bibliotecas, módulos, clases, etcétera. Ahora la novedad es que esta modularidad ya viene preconcebida, y generalmente se cuenta con gran parte del trabajo hecho gracias a bibliotecas que se suelen agrupar en función del ámbito de la aplicación que resuelven. Se encuentran entre otros: la abstracción de acceso a datos persistentes (mediante *DataObjects* o Registros Activos), los de enrutamiento de peticiones o gestores de controladores y acciones, los encapsuladores de *WebServices* o *APIs* y los generadores de *JavaScript* que tan útiles resultan para introducir *AJAX (Asynchronous JavaScript And XML)* en las aplicaciones.

3. **Gestión de múltiples entornos.** Esto consiste en poder trabajar con entornos de desarrollo, pruebas y producción de la forma más sencilla posible, y sin para ello tener que replicar constantemente la plataforma en diversos servidores. Los *frameworks*, en la mayoría de los casos, dan soporte multientorno a través de una sencilla configuración de *environments* (entornos).

4. **Herramientas de pruebas (*test*).** Supone una forma de evitar problemas de escalabilidad de una aplicación o afrontar *refactorizaciones* con ciertas garantías de éxito. Los tests/pruebas son comprobaciones en forma de expresiones lógicas de código, que deben cumplirse siempre para poder considerar que la aplicación es consistente y está funcionando de forma adecuada o al menos tal y como se espera que lo haga.

5. **Internacionalización y localización.** Casi todos los *frameworks* se han esforzado en este campo. Si se necesita una aplicación multilenguaje y adaptada a localismos tales como divisas, medidas, formatos de fecha y hora, etcétera, en casi cualquier caso es posible contar con algún soporte nativo ofrecido desde el propio *framework*.

Actualmente existe una amplia oferta de opciones ¿cuál de estas opciones elegir? Una variable muy importante a la hora de elegir un *framework* es el lenguaje de programación en el que éste se apoya. También la comunidad de desarrolladores ya establecida es muy importante a la hora de búsqueda de información o ayuda para el desarrollo. Para resumir las características de distintos *frameworks* se muestra una breve comparativa de las funcionalidades básicas de los mismos en la tabla 2.1[12].

Antes de comentar a detallar los *frameworks* evaluados para la implementación de este proyecto se detalla una tendencia actual y de gran importancia en el desarrollo de software y de aplicaciones web que es el «*desarrollo web ágil*».

2.2.1. Desarrollo Web Ágil

El «Desarrollo Web Ágil» es un paradigma de desarrollo de software basado en métodos ágiles. Este paradigma intenta evitar los obstáculos burocráticos de las metodologías tradicionales enfocándose en la gente y los resultados.

[12]Información obtenida de la revista PCActual a 26 de Noviembre de 2007

framework	Características							
	lenguaje	licencia	Google Hits	Soporte MVC	Soporte AJAX	Soporte Tests	Soporte Internacionalización	Sistema de Plantillas
Ruby on Rails	Ruby	MIT	43.8	ActiveRecord	Si	Si	Si	Si
Django	Python	BSD	17.4	n.d.	Si	n.d.	Si	Si
CakePHP	PHP	MIT	6.2	ActiveRecord	Si	Si	Si	Si
Symphony	PHP	MIT	5.6	n.d.	Si	n.d.	Si	n.d.
TurboGears	Python	MIT/LPGL	2.9	n.d.	Si	n.d.	Si	n.d.
Apache Struts	Java	Apache	2.8	n.d.	Si	n.d.	Si	n.d.
Grails	Groovy/Java	Apache	2.2	Si	Si	Si	Si	Si
Catalyst	Perl	GPL	2.1	n.d.	Si	n.d.	Si	n.d.
Seaside	SmallTalk	MIT	1.7	n.d.	Si	n.d.	Si	n.d.

Tabla 2.1: Resumen características diferentes frameworks desarrollo web

En Marzo de 2001 diecisiete críticos de los modelos tradicionales de desarrollo de software acuñaron el término «Métodos Ágiles» para definir a los métodos que estaban surgiendo como alternativa a las metodologías formales a las que consideraban excesivamente pesadas y rígidas por su carácter normativo y fuerte dependencia de planificaciones detalladas previas al desarrollo.

Los integrantes de la reunión resumieron los principios sobre los que se basan los métodos alternativos en cuatro postulados, lo que ha quedado denominado como Manifiesto Ágil[13].

- Valorar más a los individuos y su interacción que a los procesos y las herramientas.
- Valorar más el software que funciona que la documentación exhaustiva.
- Valorar más la colaboración con el cliente que la negociación contractual.
- Valorar más la respuesta al cambio que el seguimiento de un plan.

Tras los cuatro postulados descritos los firmantes redactaron los siguientes principios que se derivan y que ellos siguen[14]:

- Nuestra mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor.
- Aceptamos requisitos cambiantes, incluso en etapas avanzadas. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
- Entregamos software frecuentemente, con una periodicidad desde un par de semanas a un par de meses, con preferencia por los periodos más cortos posibles.
- Los responsables de negocio y los desarrolladores deben trabajar juntos diariamente a lo largo del proyecto.
- Construimos proyectos con profesionales motivados. Dándoles el entorno y soporte que necesitan, y confiando en ellos para que realicen el trabajo.
- El método más eficiente y efectivo de comunicar la información a un equipo de desarrollo y entre los miembros del mismo es la conversación cara a cara.
- Software que funciona es la principal medida de progreso.
- Los procesos ágiles promueven el desarrollo sostenible. Espónsores, desarrolladores y usuarios deben ser capaces de mantener un ritmo constante de forma indefinida.

[13] Puede consultarse en la dirección http://agilemanifesto.org
[14] Pueden consultarse en la dirección http://agilemanifesto.org/principles.html

- La atención continua a la excelencia técnica y los buenos diseños mejoran la agilidad.
- Simplicidad, el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- Las mejores arquitecturas, requisitos y diseños surgen de equipos que se autoorganizan.
- A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo, entonces mejora y ajusta su comportamiento de acuerdo a sus conclusiones.

Los métodos ágiles más empleados y conocidos son: *Extreme Programming* (XP), *Scrum*, *Adaptative Software Development* (ASD), *Feature Driven Development* y *Lean Software Development*.

Después de la explicación formal de lo que significa el desarrollo de software ágil podemos resumir que el desarrollo ágil es una filosofía enfocada en actuar, más que en pensar o planificar. Además es una filosofía muy utilizado en los últimos años en multitud de proyectos web.

En este proyecto se han aplicado parte de los principios que sugirieron los firmantes del manifiesto ágil. En el mundo del desarrollo web todo cambia muy rápido y lo que no existe ahora puede existir en el instante siguiente. Esta metodología promueve que una vez se tenga una aplicación que funcione técnicamente de manera aceptable se haga publica y se compruebe su funcionamiento y aceptación.

En metodologías tradicionales rectificar puede ser percibido negativamente, significa que se ha cometido un error, pero también puede ser percibido positivamente como que se ha aprendido y se es capaz de reaccionar rápidamente para arreglarlo.

A la hora de iniciar este proyecto fin de carrera, en Octubre de 2007, se estudió la conveniencia de la utilización de dos de los *frameworks* indicados en la sección anterior: **Grails** y **Ruby on Rails**. El primero de ellos por su novedad y debido a que está basado en Java (lenguaje de programación de éxito en la última década) y el segundo de ellos debido a su gran popularidad y uso en los últimos años. Ambos *frameworks* son abanderados de la metodología ágil y cada vez tienen mayor número de adeptos. Como quedará de manifiesto en los siguientes apartados sus características casan perfectamente con dicha metodología

2.2.2. Grails

Grails[15] es un *framework* para desarrollo de aplicaciones web que integra un gran número de aplicaciones web y bibliotecas Java recientes en una sola aplicación. Entre ellas, las que se muestran en la siguiente imagen[16].

Figura 2.5: Aplicaciones Java integradas en Grails

Sin embargo, aunque Grails hace uso de desarrollos en Java, está basado en el lenguaje de programación Groovy[17]. Con este lenguaje Grails se apoya en las características dinámicas que aporta Groovy.

[15]Grails http://grails.org
[16]Imagen obtenida de http://davisworld.org
[17]Groovy http://groovy.codehaus.org

Esta elección es debida a que si se parte de un conocimiento de Java, la curva de aprendizaje es prácticamente plana, ya que Groovy está basado igualmente en Java. El desarrollo en Grails puede llevarse a cabo tanto en Groovy como en Java, con pequeñas modificaciones, y los desarrolladores Java pronto podrían sacarle el máximo partido a las características dinámicas del lenguaje Groovy, manteniendo la posibilidad de utilizar la gran cantidad de aplicaciones Java ya desarrolladas.

Desde el punto de vista del diseño, Grails se basa en los principios *"Convención mejor que configuración"* (CoC (Convention over Configuration)[18]) y *"No te repitas"* (DRY (Don't Repeat Yourself) [19]). Además, trata siempre de hacer lo más sencilla posible la experiencia del desarrollador, con un ciclo de desarrollo muy ágil que permite centrarse en las funcionalidades en lugar de los requisitos técnicos del *framework*. Siempre que resulta posible se descubren las características del proyecto en tiempo de ejecución, ahorrando mucha configuración y trabajo previo no relacionado directamente con la aplicación que se quiere desarrollar.

A la hora de iniciar este estudio la versión estable de Grails era la versión 0.6, lo que pone de manifiesto la juventud de este *framework*. Aunque la comunidad lo ha acogido bien y con grandes expectativas, los distintos tutoriales y ejemplos seguidos para entender el funcionamiento y complejidad de los desarrollos en Grails demuestran que es necesario un mayor desarrollo del mismo, y que es demasiado temprano para decantarse por desarrollar la aplicación web con Grails, ya que se producen bastante errores a la hora de ejecutar incluso las aplicaciones de prueba más sencillas.

Para ampliar información se recomienda la consulta de las siguientes referencias bibliográficas: [10], [11] y [12].

2.2.3. Ruby on Rails

Anteriormente se comentó que una variable importante a la hora de elegir un *framework* es el lenguaje de programación en el que está basado. Sin embargo, resulta extraño que el *framework* de desarrollo web que mayores honores en cuanto a modelo a seguir ha despertado, Ruby on Rails[20], se base en un lenguaje bastante desconocido como Ruby[21], obviando otros lenguajes más extendidos y conocidos como PHP (PHP Hypertext Pre-processor) o Java, u otros más cercanos a Ruby en cuanto a filosofía como Python o Perl. Las aplicaciones desarrolladas con este *framework* son conformes al paradigma Modelo Vista Controlador (MVC). Paradigma cuya implementación dentro del *framework* Rails se explica en el apartado 2.2.4.

Ruby es un lenguaje de programación reflexivo, dinámico y orientado a objetos. Su sintaxis está inspirada en Perl, con características de orientación a objetos similares a las de otro lenguaje de programación como Smalltalk. También comparte otras características con Python, Lisp, Dylan y CLU. Ruby es un lenguaje interpretado de una única pasada. Su implementación oficial está licenciada como software libre y está escrita en C.

El lenguaje Ruby fue creado por Yukihiro "Matz" Matsumoto, quien empezó a trabajar en Ruby en Febrero de 1993, y lanzó la primera versión pública en 1995. Según su creador, Ruby fue diseñado para aumentar la productividad y la diversión de los programadores. En Noviembre de 2007, la última versión es la 1.8.6.

Como ya hemos comentado Ruby on Rails es un *framework* para desarrollo de aplicaciones web. Su principal objetivo es incrementar la velocidad y facilitar el manejo de la base de datos por parte de las aplicaciones web. También ofrece esqueletos de código (conocidos como *scaffolding*).

[18] Convención sobre Configuración - http://en.wikipedia.org/wiki/Convention_over_Configuration
[19] No te Repitas - http://en.wikipedia.org/wiki/Don't_repeat_yourself
[20] Ruby On Rails - http://www.rubyonrails.org
[21] Lenguaje Ruby - http://www.ruby-lang.org/es

Filosofía

Los principios fundamentales sobre los que se apoya Ruby on Rails (RoR) son los mismos que Grails, es decir, CoC y DRY.

Historia

Ruby on Rails fue creado por David Heinemeier Hansson (DHH) para su herramienta de gestión de proyectos Basecamp[22] en la compañía 37signals. Se hizo público en Julio de 2004.

En Agosto de 2006 se anunció que Apple integraría Ruby on Rails en Mac OS X v10.5 Leopard, que ha visto finalmente la luz en Octubre de 2007.

En Junio de 2008 la última versión estable de Rails es la 2.1. Aunque al inicio de este estudio se trabajó con la versión Rails 1.2.5, posteriormente la aplicación fue migrada a la versión 2.0.2 en Enero de 2008, ya que el paso a esta versión era altamente recomendable. En Junio de 2008 se migró a la versión 2.1, ya que se introducían pequeñas mejoras y se corregían errores, sin que la aplicación tuviera que sufrir cambios sustanciales.

Arquitectura

Los componentes de la arquitectura MVC son los siguientes:

1. **Modelo**. En las aplicaciones web orientadas a objetos sobre bases de datos, el *Modelo* consiste en las clases que representan a las tablas de la base de datos. Estas clases son gestionadas por la biblioteca ActiveRecord.

2. **Vista**. La lógica de visualización, o cómo se muestran los datos de las clases del *Controlador*.

3. **Controlador**. En MVC, las clases del *Controlador* responden a la interacción del usuario e invocan a la lógica de la aplicación, que a su vez manipula los datos de las clases del *Modelo* y muestra los resultados usando las *Vistas*. En las aplicaciones web basadas en MVC, los métodos del controlador son invocados por el usuario usando el navegador web.

La imagen 2.6[23] muestra cómo se realiza el procesamiento de una petición en una aplicación que sigue el patrón de diseño MVC.

Este procesamiento sigue los siguientes pasos:

1. El navegador, en el cliente, envía una petición al controlador en el servidor.

2. El controlador obtiene los datos que necesita del modelo para generar la respuesta a la petición.

3. El controlador crea la página y se la envía a la vista.

4. La vista envía la página al cliente para que el navegador la muestre.

Otras de las herramientas de desarrollo que incluye la instalación de Rails son: WEBrick (un servidor web) y Rake (ayuda a la creación de la aplicación). En entornos de producción se recomienda utilizar Apache o lighttpd como servidor web.

El acceso a la base de datos es totalmente abstracto desde el punto de vista del programador, y Rails gestiona los accesos a la base de datos automáticamente (aunque, si se necesita, se pueden hacer consultas directas en SQL (Structured Query Language)). Rails intenta mantener la neutralidad con respecto

[22] Basecamp - http://www.basecamphq.com
[23] Obtenida de [13, pág. 95]

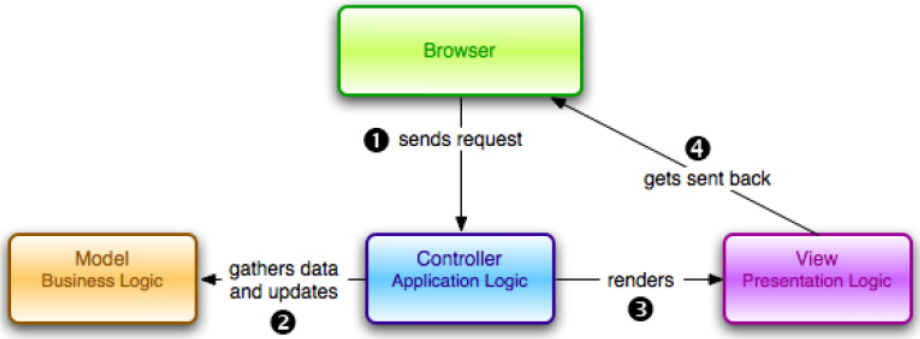

Figura 2.6: Procesamiento de una petición en una arquitectura MVC

a la base de datos, la portabilidad de la aplicación a diferentes sistemas de base de datos y la reutilización de bases de datos preexistentes. Se soportan entre otros los siguientes sistemas gestores de bases de datos: MySQL, PostgreSQL, SQLite, IBM DB2 y Oracle entre otros.

Para ampliar información se recomienda también la consulta de las siguientes citas bibliográficas: [14], [15] , [16], [17] y [18].

2.2.4. Implementación paradigma MVC en el *framework* Rails

En este apartado se exponen los detalles técnicos más relevantes de la implementación del paradigma Modelo Vista Controlador (MVC) en el *framework* de desarrollo web Rails. La figura 2.7[24] representa el flujo desde que se realiza una petición desde un navegador hasta que se devuelve la respuesta a dicha petición utilizando Rails.

En el anterior flujo quedan de manifiesto las diferentes partes que componen cada una de las partes del patrón MVC y que clases del *framework* son las encargadas de parte de la implementación:

- **Controlador.** Se implementa con la clase **ActionController**.
- **Modelo.** Se implementa con la clase **ActiveRecord**.
- **Vista.** Se implementa con la clase **ActionView**.

A continuación se detalla la funcionalidad que proporcionan cada unas de las clases que permiten la implementación del paradigma MVC en Rails.

Controlador - ActionController

Como cualquier otro programa las aplicaciones Rails siguen un flujo de una parte de la aplicación a otra. La parte fundamental del *framework* que gestiona este flujo son los controladores.

Una buena práctica es evitar situar la lógica de la aplicación en los controladores y situarla en los modelos. Los controladores solo deben gestionar el control de acceso, recuperar información de la base de datos y hacer que este disponible en las vistas.

Cuando alguien se conecta a la aplicación lo que se esta haciendo es invocar la ejecución de una acción de un determinado controlador. Como en toda aplicación web en las aplicaciones Rails un servidor web manejará las peticiones (Apache, Lighttpd, Mongrel). El servidor entregará dicha petición a

[24]Imagen perteneciente al documento ¿Qué es Ruby on Rails?

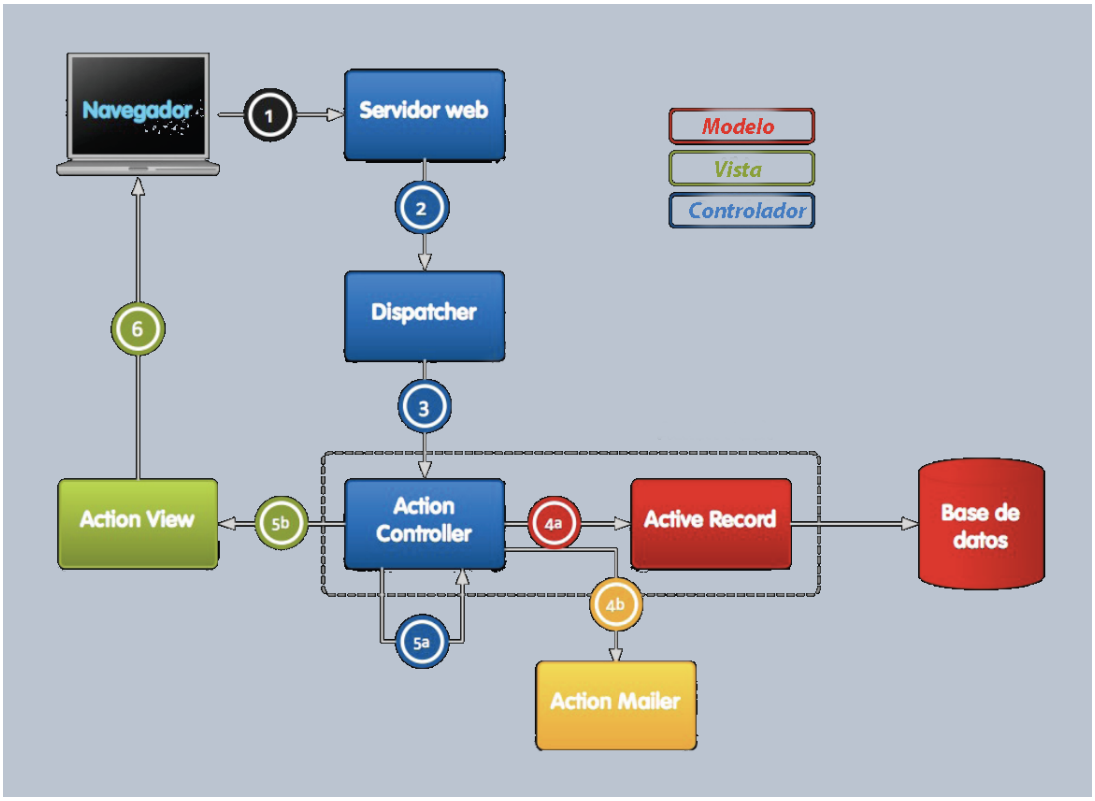

Figura 2.7: Flujo patrón MVC en Ruby on Rails

la aplicación donde será manejada por el planificador *dispatcher*[25] en primer lugar. El *dispatcher* será el encargado de averiguar a que controlador dirigir la petición y que acción ejecutar de dicho controlador. La especificación del mapeo entre URLs y el controlador y acción se especifica en el fichero de rutas[26] (alojado en `config/routes.rb`).

Uno de los objetivos principales de los controladores es *construir* las plantillas, es decir, crear el contenido (generalmente HTML) a partir de las variables y la propia plantilla, y pasarlo al servidor web que lo entregará al usuario.

Si existe el siguiente código de controlador:

```
class TracksController < ApplicationController
  def index
  end
end
```

Si no se específica ninguna plantilla a *construir* (*"render"*) en el controlador se asume el comportamiento por defecto, y este es *construir* la plantilla que se corresponde con el nombre del controlador y la acción, en este caso `app/views/tracks/index.html.erb`. Por lo tanto, las acciones incluyen el comando `render` implícitamente. Esta es una de las convenciones del *framework*.

También es posible especificar la plantilla a utilizar o especificar otra acción a *construir*. Pero aquí se debe aclarar que no se ejecuta la acción sino que solo se ejecuta el comando `render` asociado a esa acción. Otras opciones de ese comando incluyen *construir*: texto (`:text`), código en línea (`:inline`), y otros tipos de datos estructurados como son JSON (JavaScript Object Notation) y XML.

[25] Se utilizará la voz inglesa dada su aceptación en el ámbito de la aplicaciones web
[26] Rutas en Rails consultar - http://guides.rails.info/routing/routing_outside_in.html

```
class TracksController < ApplicationController
  def index
    if format == :json
      render :json => @record.to_json
    elsif format == :xml
      render :xml => @record.to_xml
    end
  end
end
```

En los casos anterior cuando se ejecuta el comando `render` se termina el ciclo de la petición. Pero en ocasiones es preferible redirigir la petición hacia otra acción. Con el método `redirect_to` se realiza una nueva petición sobre otra acción y se comienza un nuevo ciclo hasta que se decide que es lo que se debe construir.

La comunicación de los controladores con las vistas se comentará a partir de la página 24.

Filtros

Otra funcionalidad importante de lo controladores en Rails son los filtros (*filters*) son pequeñas porciones compartidas de código de procesamiento que se ejecutan antes (*before*) o después (*after*) de las acciones. Los filtros se utilizan generalmente para autenticación, caché, o como control de acceso antes de ejecutar una determinada acción.

El siguiente código muestra un ejemplo de su uso:

```
class TracksController < ApplicationController

  before_filter :login_required

  def index
    if format == :json
      render :json => @record.to_json
    elsif format == :xml
      render :xml => @record.to_xml
    end
  end
  ...

  private
  def login_required
  # comprueba que el usuario inicio una sesión
  end
end
```

En el anterior ejemplo el filtro se ejecutará antes de invoacar al método de cada una de las acciones. Es posible especificar el conjunto de acciones sobre las que se debe ejecutar dicho filtro (opción `:only =>[:accion1, :accion2]`) o especificar las acciones sobre las que no se debe ejecutar (opción `:except =>[:accion3, :accion4]`).

Aquí solo se han cubierto unos pocos puntos relevantes de los controladores en Rails. Por lo tanto, se aconseja al usuario consultar la documentación sobre ActionController[27] para profundizar en el tema.

Modelo - ActiveRecord

ActiveRecord toma su nombre del patrón de diseño ActiveRecord identificado por Martin Fowler en su trabajo *"Patterns of Enterprise Architecture"*. En este patrón se mapea una clase en una tabla de la base de datos y cada instancia de la clase se mapea con una fila de la tabla de la base de datos.

[27] ActionController - http://api.rubyonrails.com/classes/ActionController/Base.html

Simplificando la definición, ActiveRecord es una biblioteca de código escrita en Ruby que permite la interacción con bases de datos y proporciona persistencia de objetos a la aplicación. Esta biblioteca cuenta con numerosos adaptadores para la utilización de muy diferentes gestores de bases de datos. Esto hace que la utilización de la biblioteca sea independiente casi en su totalidad del gestor de base de datos que se utilice.

ActiveRecord incluye mecanismos para representar modelos y sus relaciones, operaciones DRY, búsquedas complejas, validaciones, *callbacks*, y muchas otras funcionalidades. Una de sus características más importantes es que sigue el principio *Convención sobre Configuración* y esto facilita enormemente su uso cuando se crea una nueva base de datos que siga la convención. Aunque también permite su configuración para la utilización en base de datos de sistemas heredados. A continuación se expone como trabaja esta biblioteca en diferentes aspectos.

Para crear una nueva clase para un modelo, la clase debe heredar de ActiveRecord::Base:

```
class Track < ActiveRecord::Base
end
```

Por convención la clase `Track` será mapeada con la tabla `tracks` de la base de datos, mediante un mecanismo de *"pluralización"* (*pluralization*). También por convención espera que la tabla tenga un columna `id` para usarla como clave primaria. Cada una de las instancias de la clase proporciona acceso a una de las filas de la tabla de base de datos correspondiente siguiendo una filosofía de orientación a objetos. Cada columna de la tabla se corresponderá con un atributo del objeto. Y para añadir, eliminar y cambiar atributos y sus tipos se realizará sobre la tabla de la base de datos y los cambios se reflejarán automáticamente en las instancias de la clase.

La definición de las tablas de base de datos y también su modificación se realiza mediante unos ficheros que se conocen como ficheros de migración[28]. Un ejemplo de fichero de migración para la creación de la clase anterior sería:

```
class CreateTracks < ActiveRecord::Migration
  def self.up
    create_table :tracks do |t|
      t.string :title
      t.timestamps
    end
  end

  def self.down
    drop_table :tracks
  end
end
```

Con la migración anterior se crearía una tabla con nombre `tracks` y con las columnas: `id`, `title`, `created_at` y `updated_at`. Las dos últimas columnas son manejadas de forma automática por ActiveRecord cuando se crea o actualiza el objeto en esos campos se asigna la fecha de dicha creación o actualización respectivamente.

Operaciones CRUD

Las operaciones básicas sobre los objetos de la base de datos se combinan en el acrónimo DRY. ActiveRecord facilita estas operaciones como se verá a continuación.

Para crear un nuevo objeto de la clase `Track` se realiza lo siguiente :

```
>>t = Track.new
#<Track:0x2515584>
```

[28] Consultar la documentación - http://api.rubyonrails.com/classes/ActiveRecord/Migration.html

devolviendo una instancia de la clase `Track`, pero que aún no esta almacenada de forma persistente. Para crear una nueva intancia almacenarla y que sea devuelta haríamos lo siguiente:

```
>> t = Track.create(:title =>"Munich")
#<Track:0x4229490>
```

Para la lectura de objetos de la base de datos ActiveRecord proporciona diferentes mecanismos. El primero de ellos son los métodos `find`. A continuación se exponen algunos ejemplos de su uso utilizando diferentes parámetros y opciones mostrando la consulta que se realiza sobre la base de datos:

```
>>Track.find(1)
SELECT * FROM 'tracks' WHERE ('tracks'.'id' = 1)
>>Track.find(1,2)
SELECT * FROM 'tracks' WHERE ('tracks'.'id' IN (1,2))
>>Track.find(:first)
SELECT * FROM tracks LIMIT 1
>>Track.find(:last)
SELECT * FROM tracks ORDER BY tracks.id DESC LIMIT 1
```

También es posible establecer condiciones sobre las consultas. Aquí algunos ejemplos:

```
>>Track.find(:all, :condtions =>["created_at >? AND created_at <?", params[:start_date], params[:end_date]])
SELECT * FROM 'tracks' WHERE ('tracks'.'created_at' >'08-10-2008' AND 'tracks'.'created_at' <'28-10-2008')
```

Establecer el orden de los registros devueltos:

```
>>Track.find(:all, :order =>"created_at DESC")
SELECT * FROM `tracks` ORDER BY `tracks`.`created_at` DESC
```

Seleccionar exclusivamente ciertos atributos:

```
>>Track.find(:first, :select =>"id")
SELECT id FROM `tracks` LIMIT 1
```

También es posible establecer límites, *offsets* y agrupar registros:

```
>>Track.find(:all, :limit =>5)
SELECT id FROM `tracks` LIMIT 5
>>Track.find(:all, :limit =>5, :offset =>5)
SELECT * FROM trackss LIMIT 5, 5
>>Track.find(:all, :group =>"title")
SELECT id FROM `tracks` GROUP BY title
```

Una característica también muy importante son los buscadores dinámicos basados en atributos (*Dynamic Finders Methods*). Son métodos de búsqueda que se crean de forma dinámica gracias al *callback* ***method_missing***[29]. Estos buscadores permiten facilitar la búsqueda de objetos mediante condiciones. Aquí un ejemplo:

```
>>Track.find_by_title("Munich")
SELECT * FROM `tracks` WHERE (`tracks`.`title` = 'Munich') LIMIT 1
```

Es posible también indicar la consulta con sintaxis SQL:

```
>>Track.find_by_sql("SELECT * FROM tracks")
SELECT * FROM `tracks`
```

Para la actualización de atributos se pueden utilizar algunos de los siguientes métodos:

[29] Más información - http://blog.hasmanythrough.com/2006/8/13/how-dynamic-finders-work

```
>>t = Track.find(:first)
>>t.update_attribute(:title, params[:title])
ó
>>t.title = params[:title]
>>t.save
```

Para eliminar un registro de la base de datos existen dos opciones. Si se cuenta con una instancia de la clase:

```
>>t = Track.find(:first)
>>t.destroy
```

Si no se cuenta con la instancia se pueden utilizar métodos de la propia clase:

```
>>Track.delete(1)
>>Track.destroy([2, 3])
```

Relaciones entre objetos

La capacidad y la sencillez en el establecimiento y manipulación de relaciones entre objetos con ActiveRecord es una de las características más especiales e importantes de esta biblioteca.

Relación 1-n Ejemplo de establecimiento de una relación 1-n ó uno a muchos.

```
class User < ActiveRecord::Base
  has_many :tracks
end
class Track < ActiveRecord::Base
  belongs :user
end
```

Cuando las clases se cargan las anteriores declaraciones son ejecutadas y Rails mediante *meteprogramación* añade diferentes métodos a los modelos que permiten manipular sencillamente las relaciones entre objetos. Establecer la anterior relación de modo tan sencillo es debido al uso de convenciones. Con `has_many :tracks` se busca una tabla en la base de datos con nombre `tracks`. Con `belongs_to :user` se busca una tabla en la base de datos con nombre `users`, la declaración de un modelo User y una columna en la tabla `tracks` de nombre `user_id`. Por lo tanto, es necesario establecer la columna que se utilizará como clave foránea en la tabla `tracks`.

```
class AddUserIdToTracks < ActiveRecord::Migration
  def self.up
    add_column :tracks, :user_id, :integer
  end

  def self.down
    remove_column :tracks, :user_id
  end
end
```

La anterior migración junto con la relación da lugar a la creación de diferentes métodos algunos de ellos son[30]:

- **user.tracks** recupera todos los registros de la asociación.
- **user.tracks** << **track** añade el objeto a la asociación.
- **user.tracks.delete(track)** elimina la instancia de la asociación.

[30]Consultar el API de Rails http://api.rubyonrails.org/

- **user.tracks.empty?** indica si existen registros en la asociación.
- **user.tracks.size** número de objetos asociados.
- **user.tracks.find** realizar búsquedas sobre los registros asociados.
- **track.user** recupera el registro asociado.
- **track.user=u** establece el registro asociado.
- **track.user?** consulta si existe el registro asociado.

Relaciones m-n Ejemplo de establecimiento de una relación m-n ó muchos a muchos[31].

A nivel de modelos:

```
class Developer < ActiveRecord::Base
  has_and_belongs_to_many :projects
end
class Project < ActiveRecord::Base
  has_and_belongs_to_many :developers
end
```

En cuanto a ficheros de migraciones

```
class CreateDevelopers < ActiveRecord::Migration
  def self.up
    create_table :developers do |t|
      t.string :name

      t.timestamps
    end
  end

  def self.down
    drop_table :developers
  end
end

class CreateProjects < ActiveRecord::Migration
  def self.up
    create_table :projects do |t|
      t.string :name

      t.timestamps
    end
  end

  def self.down
    drop_table :projects
  end
end

class CreateDevelopersProjectsTable < ActiveRecord::Migration
  def self.up
    create_table :developers_projects, :id => false do |t|
      t.integer :developer_id
      t.integer :project_id
      t.timestamps
    end
  end

  def self.down
    drop_table :developers_projects
  end
end
```

[31] Ejemplos tomados del API de Rails

Para que la relación muchos a muchos se realice a través de una tabla *join* de un modo simple, es necesario, como se dijo anteriormente seguir las convenciones. En este caso, el método `has_and_belongs_to_many` busca una tabla en la base de datos con nombre `developers_projects`. Los nombres en plural de los modelos se concatenan en orden alfabético. También es necesario que esa tabla presente al menos las columnas: `developer_id` y `project_id` que serán las que se utilizarán como claves para la relación muchos a muchos de los modelos.

Los métodos que se generan en el modelo `Developer` son:

- **developer.projects** recupera todos los registros de la asociación.
- **developer.projects** << **project** añade el objeto a la asociación.
- **developer.projects.delete(project)** elimina la instancia de la asociación.
- **developer.projects.empty?** indica si existen registros en la asociación.
- **developer.projects.size** número de objetos asociados.
- **developer.projects.find(id)** realizar búsquedas sobre los registros asociados.

Los mismo métodos se generan en el modelo `Project`. Es posible obtener otro tipo de asociaciones entre los modelos, por lo que se recomienda consultar la guía sobre asociaciones en Rails[32] para ampliar información.

Validaciones

Las validaciones que proporciona ActiveRecord permiten definir de forma declarativa las validaciones de los modelos. A continuación se muestran algunas de las validaciones que se pueden establecer sobre un modelo:

validates_confirmation_of :password Permite confirmar que los datos críticos introducidos por el usuario en dos campos coinciden.

validates_inclusion_of :gender, :in =>['m','f'] Valida que el valor para el campo gender es 'm' o 'f'.

validates_format_of :email, :with =>/\A([@\s +)@((?:[-a-z0-9]+\.)+[a-z]2,)\Z/i] Valida el valor del campo email contra una expresión regular.

validates_length_of :login, :minimum =>5 Establece que los valores para el campo login deben tener una longitud de al menos 5 carácteres.

validates_numericality_of :account_number Establece que los valores válidos del campo account_number deben ser de tipo numérico.

Prácticamente todas las validaciones presentan diferentes opciones que permiten: decidir bajo que condición se ejecutan (`:if`), el mensaje devuelto cuando no se cumple la validación (`:message`), decidir cuando se ejecutan la validaciones (`:on`). También es posible definir métodos de validación propios aunque muy probablemente no será necesario y será suficiente con los métodos ya disponibles.

[32] Asociaciones en Rails - http://guides.rails.info/activerecord/association_basics.html

Callbacks

Los *callbacks* son porciones de código que se ejecutan antes o después de que se produzcan modificaciones en el estado de un objeto. Estos métodos son muy útiles para realizar determinadas operaciones cuando se produce una determinada modificación en el estado de un objeto. Los *callbacks* disponibles son: `after_create`, `after_destroy`, `after_save`, `after_update`, `after_validation`, `after_validation_on_create`, `after_validation_on_update`, `before_create`, `before_destroy`, `before_save`, `before_update`, `before_validation`, `before_validation_on_create` y `before_validation_on_update`.

Para utilizarlos:

```
class User < ActiveRecord::Base
  callback :execute_method

  private
  def execute_method
    ...
  end
end
```

Se especifica el *callback* a utilizar y el método que se ejecutará como reacción a ese cambio de estado.

Vista - ActionView

Las vistas de una aplicación constituyen la parte visible de la misma que llega a los usuarios. ActionView es la biblioteca de clases que establece el componente visual de la aplicación. Es la encargada entre otras cosas de mostrar en los navegadores el HTML asociado de la aplicación. Ahora con la adopción del paradigma REST ActionView generará cualquier tipo de formato de salida que especifiquen los controladores de la aplicación de entre los formatos disponibles, entre otros: JSON y XML.

ActionView utiliza un sistema de plantillas basada en la biblioteca de código Ruby denominada ERb[33]. Este sistema toma los datos preparados por el controlador y los intercala con el código de las vistas para que sea enviado por el controlador al servidor web. En las vistas los desarrolladores incluyen por tanto código ERb, muy similar al código que se utiliza en las páginas JSP (Java Server Pages).

Para el tratamiento de las vistas Rails sigue ciertas convenciones. Dentro del directorio app/views de la aplicación se encontrará un directorio con el nombre correspondiente a cada uno de los controladores. Dentro de uno de estos directorios se encontrarán las plantillas asociadas con cada una de sus acciones. Los nombres y tipos de fichero de estas plantillas se construyen del siguiente modo: **nombre_accion.formato.interprete**. Algunos ejemplos de esta nomenclatura son:

- **index.html.erb** Se mostrará para formatos de petición HTML con interprete ERb para la acción index. Si no se modifica el comportamiento por defecto.
- **index.js.rjs** Se mostrará para formatos de petición JavaScript (JS) con el interprete Ruby JavaScript (RJS) para la acción index. Si no se modifica el comportamiento por defecto.

Esta nomeclatura será la que se tome por defecto para buscar la plantilla a utilizar aunque se puede forzar a que una acción utilice una plantilla, formato e interprete determinadas.

Para el caso de la estructura de la vista o *layout* existe en el directorio app/views un directorio con nombre layouts. En este directorio se sitúan los *layouts*. Se debe tener en cuenta la jerarquía de controladores, todos los controladores descienden del ApplicationController. La búsqueda del *layout* se realiza del siguiente modo: si en el directorio *layouts* existe un fichero con el nombre del controlador se utilizará

[33] Documentación ERb - http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc/

por defecto ese *layout* en el caso de que no exista se buscará otro *layout*, más general, en orden jerárquico llegando al *layout* application. Igualmente se puede evitar el comportamiento por defecto mediante la invocación de la utilización de un determinado *layout*.

Otro de los componentes importantes a la hora de estructurar las vistas de una aplicación son las denominadas *partials*. Las *partials* son pequeñas porciones de código de una plantilla utilizadas para simplificar el código de las mismas. Los nombres de los ficheros de *partials* deben comenzar con un guión bajo: _track.html.erb. Para utilizar una *partial* en una plantilla bastaría con:

```
<%= render :partial => "nombre" %>
```

Donde nombre se refiere al nombre de la *partial* a partir del guión bajo.

Las *partials* se utilizan para reutilizar código en las plantillas de diferentes acciones. También es posible compartirlas entre diferentes controladores pero para ello es necesario incluir la estructura de directorios para la búsqueda de la *partial*. Por ejemplo:

```
<%= render :partial => "compartido/nombre" %>
```

Se buscaría la *partial* app/views/compartido/_nombre.html.erb .

La siguiente figura 2.8[34] ejemplifica lo explicado hasta ahora de la estructuración de las vistas.

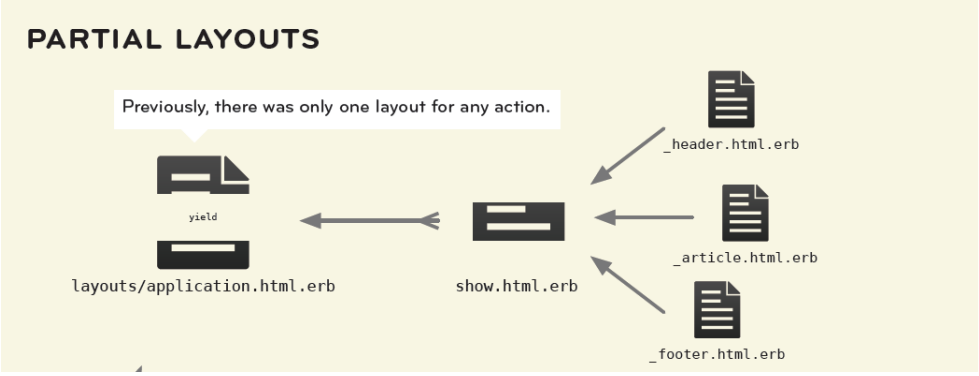

Figura 2.8: *Layouts*, plantillas y *partials* en Rails

Se ha comentado el funcionamiento de las vistas pero no que datos son los que mostrarán. Los datos a mostrar serán las variables de instancia declaradas en el controlador:

```
class WelcomeController < ActionController::Base
  def index
    @msg = ''Hello!''
  end
end
# template file /app/views/welcome/index.html.erb
<%= @msg %>
```

A las *partials* también es posible pasarle variables. Existen bastantes convenciones más que se siguen en la utilización de *partials* que no se comentarán para no extender excesivamente el documento[35].

Rails proporciona un conjunto de módulos de funciones denominadas *helpers* que facilitan la creación de la interfaz de las vistas. Entre otras cosas fácilitan: creación automática de formularios, reporte de errores de validación, métodos para enlazar documentos HTML con otro tipo de recursos (imágenes, hojas de estilos, bibliotecas JavaScript), manejo de fechas, métodos para la inclusión de JavaScript en el documento HTML, métodos de paginación, identificación de objetos. También es posible escribir estos *helpers* para obtener un código más limpio en las vistas.

[34] Imagen obtenida de http://peepcode.com/products/rails-2-pdf
[35] Guía sobre *partials* - http://guides.rails.info/actionview/layouts_and_rendering.html

2.3. Información Musical en la Red

En este punto se resumen las diferentes fuentes de información musical que son accesibles a través de Internet, al menos las que se consideran más importantes. Entre ellas cabe destacar Last.fm[36], MusicBrainz[37] y Discogs[38]; las dos últimas enfocadas directamente como base de datos de contenido musical y Last.fm enfocada más hacia la creación de una gran red social de contenido musical.

También dentro de los sitios web que ofrecen información musical cabe destacar los orientados exclusivamente a proporcionar las letras de las canciones. Entre ellos cabe destacar LyrickWiki[39]. Se destaca este sitio web fundamentalmente porque la información que almacena puede ser consultada mediante servicios web. En el caso de conseguir información acerca de la letra de las canciones existen *plugins* para el reproductor de música Amarok que lo que hacen es «*parsear*» páginas web de diferentes portales de búsqueda de letra de canciones. Estos *parsers* están escritos en Ruby, por lo que su integración sería sencilla en la aplicación a desarrollar, pero se ha considerado mejor opción la consulta a través de servicios web ya que el objetivo de estos servicios es la interoperabilidad entre aplicaciones.

En los últimos años, dado el avance de la banda ancha, han surgido sitios web que alojan vídeos de contenido musical como son: YouTube[40] y Yahoo! Music[41].

2.3.1. Last.fm

Last.fm es el sitio web de mayor éxito dentro de lo que se denominan redes sociales dedicadas a la información musical. Básicamente, es una radio vía Internet y un sistema de recomendación musical en el que se construyen perfiles y estadísticas sobre gustos musicales.

Los datos musicales se envían a la base de datos de Last.fm (*'Scrobbled'*) provenientes de los usuarios registrados. Para ello los usuarios tienen dos opciones: bien utilizar la radio de last.fm o descargarse e instalar un plugin para su reproductor de música favorito, o utilizar el propio reproductor de Last.fm. Mientras el usuario utiliza normalmente su reproductor, la información relativa a la música que escucha, como el nombre de la canción, el artista, etcétera. es enviada a su cuenta de usuario en Last.fm. Este sitio web ofrece gran cantidad de servicios de una red social como la relación entre usuarios con gustos musicales similares.

Actualmente, Last.fm es el conjunto de dos aplicaciones generales: Last.fm y Audioscrobbler, que se fusionaron en 2005.

Características

En Last.fm existen dos tipos de cuentas de usuario: las gratuitas y las de pago, cuyas características resumimos a continuación:

Cuentas Gratuitas (Free User Accounts)

Una cuenta de usuario gratuita permite el acceso a los servicios que se listan a continuación. Los usuarios registrados también tienen la posibilidad de escribir en el foro, recibir y enviar mensajes priva-

[36] Last.fm - http://www.lastfm.com
[37] MusicBrainz - http://musicbrainz.org
[38] Discogs - http://www.discogs.com
[39] LyricWiki - http://www.lyricwiki.org
[40] YouTube - http://www.youtube.com
[41] Yahoo! Music - http://music.yahoo.com

dos al resto de usuarios y también presentan la posibilidad de utilizar el reproductor de Last.fm en su máquina local.

- **Perfiles.** Cada usuario puede construir su perfil musical de muy diferentes maneras: escuchando la radio que incorpora el sitio web de Last.fm, instalando un *plugin* en su reproductor habitual, el cual envía la información de la música que escucha el usuario, etcétera. Con toda esta información se generan las recomendaciones musicales.

  Last.fm también genera una página personal con información personal que pueden introducir el usuario como: fotografías, puede modificar el *layout* de la página, también es posible incluir listas de amigos, etiquetas favoritas, grupo o eventos. Crear listas de reproducción propias.

- **Recomendaciones.** Las recomendaciones son uno de los servicios más recientemente renovado en Last.fm se incluye en una página personal denominada *"The Dashboard"*, sólo disponible para el usuario que se autentica en la aplicación. En esta página se sugiere nueva música, eventos y otras personas con gustos musicales similares. Last.fm también permite a los usuarios recomendar de forma manual canciones, artistas o álbumes específicos a otros usuarios a sus listas de amigos o grupos a los que pertenezca el usuario.

- **Grupos.** Esta característica es quizá la más utilizada dentro de la comunidad de usuarios. Permite la formación de grupos de usuarios entre usuarios con algo en común (por ejemplo, fans de un determinado artista, o género musical, etcétera). Last.fm genera un perfil de grupo muy parecido al perfil de usuario. También se genera una estación de radio de grupo basada en los perfiles de los usuarios que forman ese grupo.

- **Eventos.** En Octubre de 2006 fue añadida esta funcionalidad que permite a los usuarios especificar lugares donde se celebrarán festivales que al usuario le gustaría ver. La información relativa del festival o concierto se añadirá a la página principal del artista. Para que sea visible por el resto de usuarios de la comunidad, también existe la posibilidad de añadir fotografía y comentarios de eventos pasados.

Cuentas de Pago (*Subscriber Accounts*)

Last.fm oferta cuentas de pago, cuyo coste varia entre £1.50 y £3.50 al mes. Algunas de las características extras que reciben los usuarios de pago son:

- No tiene publicidad
- Más opciones de configuración en la estación de radio
- Poder ver quien ha visitado nuestro perfil.
- Acceso a las pruebas beta
- Cambio del color del icono de usuario.

Para ampliar información sobre esta aplicación acceder a Wikipedia/Last.fm[42] o simplemente acceder a Last.fm[43] y utilizar el servicio, el cual está disponible también en español (Lastfm.es[44]).

[42] Wikipedia/Last.fm - http://en.wikipedia.org/wiki/Last.fm
[43] Last.fm - http://www.lastfm.com/
[44] Last.fm en español http://www.lastfm.com/

Audioscrobbler

Last.fm, como ya se ha comentado, es un proyecto unido a AudioScrobbler. Audioscrobbler es la base de datos que se encuentra detrás del sitio web Last.fm.

El sistema Audioscrobbler es una base de datos de hábitos de escucha de canciones. Además, calcula relaciones y recomendaciones basadas en lo que escuchan los distintos usuarios.

Parte de todos los datos que almacena este sistema son accesibles vía servicios web[45], de este modo otros proyectos pueden crear servicios a partir de los datos y recomendaciones que proporciona. En el capítulo 4 se explicarán las características de este API y el modo en que se pueden obtener los datos del mismo.

Este API va a ser una pieza fundamental en esta aplicación ya que proveerá gran cantidad de contenido a los usuarios de la aplicación.

2.3.2. MusicBrainz

MusicBrainz[46] es un proyecto de la fundación estadounidense sin ánimo de lucro MetaBrainz, que pretende crear una base de datos musical de contenido abierto. Con esta base de datos musical se intenta crear un sitio web de información musical. Los datos contenidos en MusicBrainz se pueden acceder navegando en el sitio web, o accediendo desde un programa cliente. También es posible utilizar el etiquetador de MusicBrainz para identificar automáticamente los metadatos de las etiquetas de una colección de música digital.

Desde la propia página de MusicBrainz los usuarios registrados pueden crear y mantener la información disponible sobre los discos de forma que, muy al estilo de la web 2.0, la base de datos va mejorando de forma colaborativa.

Inicialmente, MusicBrainz utilizó el algoritmo TRM (un acrónimo recursivo que significa: TRM *Recognizes Music*; TRM Reconoce Música) de *Relatable* para la búsqueda de coincidencias mediante el uso de una huella digital acústica; un código único generado que permite identificar cada una de las pistas. Esta característica, atrajo muchos usuarios y permitió que la base de datos creciera de forma muy rápida. En 2005 fue obvio que el método de huellas digitales de Relatable no escalaba bien para los millones de pistas que, en ese entonces, había en la base de datos por lo que comenzó la búsqueda de una alternativa viable.

El 12 de marzo de 2006, Robert Kaye escribió un anuncio en el blog oficial del proyecto acerca de un acuerdo entre MusicBrainz y MusicIP[47]. Parte del acuerdo incluía el permitir a MusicBrainz el uso del servicio MusicDNS[48] de huellas digitales acústicas (PUIDs) de MusicIP. Después de un período de transición de 6 meses, los códigos TRM fueron eliminados y MusicBrainz ha pasado a utilizar solamente PUIDs.

[45] Servicios Web Audioscrobbler - http://www.audioscrobbler.net/data/webservices/
[46] MusicBrainz - http://musicbrainz.org
[47] MusicIP - http://www.musicip.com/
[48] MusicDNS - http://www.musicip.com/dns/index.jsp

Los datos que alberga MusicBrainz (artistas, pistas, álbumes, etcétera) son de dominio público, y el contenido adicional incluyendo tablas generadas en la búsqueda, anotaciones, estadísticas y ediciones se publican bajo la licencia Creative Commons ShareAlike 2.0.

2.3.3. Discogs

Discogs[49] es otra base de datos y sitio web que contiene información musical. Desde mediados de Agosto de 2007 Discogs dispone de una API accesible mediante REST, lo que permite recuperar información musical de su amplia base de datos.

La información que se suministra a la página de Discogs solo puede ser realizada por usuarios registrados en el sitio web. La información que se pretende agregar primero debe ser evaluada por un grupo de *Moderadores* quienes someten a votación la información para evitar errores, duplicados o vandalismos.

2.3.4. LyricWiki

LyricWiki.org[50] es un sitio web orientado al almacenamiento de letras (*lyrics*) de canciones. Está construido mediante un servidor de *wikis*. El propósito del sitio web es ser una fuente donde cualquiera puede recuperar letras de cualquier canción, cualquier artista sin que se le invada con un montón de publicidad.

Este sitio web fue creado en Abril de 2006. Lyricwiki ha tenido una gran cobertura alrededor del mundo incluyendo artículos en importantes *blogs*.

Algunas de sus características son:

- 460.000 páginas de contenido.
- *"Song of the Day" & "Album of the Week"*.
- Originalmente no tenía *banners* ni publicidad, pero a partir de Abril de 2007 se vieron obligados a incluirla por problemas de financiación.
- Feed del iTunes Music Store de las canciones más populares.
- Servicio Web accesible mediante SOAP.
- Actualmente está en desarrollo una alternativa de acceso al servicio mediante REST.
- *"LyricWiki challenge"* aplicación para Facebook.

En el capítulo 4 explicamos como obtener información a través de SOAP[51] de este sitio web.

[49] Discogs - http://www.discogs.com/
[50] Lyricwiki.org - http://www.lyricwiki.org/Main_Page
[51] Información sobre SOAP consultar http://es.wikipedia.org/wiki/SOAP

2.3.5. Vídeos Musicales

Otra de las grandes fuentes de información de contenido musical son los sitios web que permiten a los usuarios compartir vídeos musicales en Internet. A continuación se resumen dos de los sitios web en los que se puede encontrar una gran cantidad de vídeos digitales de contenido musical. Se pueden encontrar desde videoclips, conciertos en directo, etcétera de los artistas preferidos o simplemente de los artistas consultados.

YouTube

YouTube[52] es una empresa fundada en febrero de 2005 y permite a sus usuarios subir y compartir videoclips de una forma sencilla en www.YouTube.com y a través de Internet mediante sitios web, dispositivos móviles, *blogs* y correo electrónico.

YouTube se ha convertido en el líder en Internet en compartir videoclips. Como ya se ha comentado, de entre todos los vídeos es posible realizar búsquedas según aficiones e intereses, en el caso de la aplicación en estudio se centrará en la búsqueda de contenido musical.

Gran parte de su éxito está basado en que es posible realizar consultas mediante un *API* al sitio web para crear nuestros propios proyectos a partir del contenido que aloja.

Yahoo! Music

Yahoo! Music[53] es un nuevo portal web, que ha visto la luz durante la realización de este trabajo, que ofrece contenido musical disponible en Internet. Yahoo! Music ofrece una amplia colección de *streaming* de audio, radio en Internet, y productos de artistas en exclusiva y noticias musicales que cubren todos los géneros musicales de los visitantes de Yahoo! Music. Según su página web Yahoo! Music aloja la mayor colección de vídeos musicales de Internet.

Entre los servicios que ofrece están:

- *Yahoo! LAUNCHcast*, un servicio de música en *streaming* que permite a los usuarios crear sus propias estaciones de radios según sus gustos musicales.
- *Music Videos*, una gran colección de vídeos de contenido musical, y además permite crear estaciones de vídeo personalizadas.
- *Music News*, últimas noticias, entrevistas e información acerca de tus artistas favoritos.
- *Original Videos and Programming*, programación y contenido exclusivo para Yahoo! Music de los principales artistas.
- *Yahoo! Music Unlimited*, servicio de Yahoo! Music que ofrece cerca de 1 millón de canciones bajo suscripción.

[52] YouTube - http://www.youtube.com
[53] Yahoo! Music - http://music.yahoo.com

- *Yahoo! Music Jukebox*, reproductor musical gratuito de Yahoo!.

Por último, comentar algunas de las aplicaciones que existen o existieron en la web que ofrecen distintos servicios relacionados con información musical como pueden ser: Anywhere.fm, MusicMobs, iLike, FoxyTunes.

2.4. Aplicaciones

A continuación mostramos algunas de las aplicaciones accesibles a través de la web de contenido enteramente musical.

2.4.1. Anywhere.fm

Anywhere.FM[54] es una radio online que permite a los usuarios subir música al sitio web. La música que cada usuario a subido está disponible en una especie de radio personal. En la página se describe la aplicación de la siguiente manera:

Anywhere.FM es un potente reproductor musical que hace fácil:

- Subir a la red toda tu colección musical.
- Escuchar tu colección en cualquier sitio en el mejor reproductor web.
- Descubrir nueva música a través de radios de amigos.

El sitio web permite subir tu música para tu uso personal. Sin embargo no soporta subir más de un fichero de música al mismo tiempo. El otro método de subir la colección de música es mediante la instalación de un software que permite a los usuarios elegir qué carpetas de música subir a la aplicación, o seleccionar las listas de reproducción de las bibliotecas de iTunes, WinAMP o Windows Media Player, o también subir toda la colección musical.

La capacidad de almacenamiento de este sitio web es proporcionada por Amazon mediante un servicio denominado S3.

2.4.2. Musicmobs

Musicmobs era un sitio web donde los usuarios después de descargarse el plugin mobster subían la información relativa a su colección musical. Este mismo plugin permitía la sincronización con la reproducción habitual de música del usuario en su reproductor local. Musicmobs utilizaba filtrado colaborativo[55] para ofrecer recomendaciones musicales a sus usuarios.

Musicmobs utilizaba de forma extensiva el formato de listas de reproducción XSPF para hacer la distribución de las mismas de forma más sencilla. En Noviembre de 2007 el sitio cerró, cuando su creador, Toby Padilla, se unió al equipo de Last.fm.

[54] Anywhere.fm - http://www.anywhere.fm
[55] Wikipedia Filtrado Colaborativo - http://en.wikipedia.org/wiki/Collaborative_filtering

2.4.3. iLike

iLike[56] es un sitio web que permite descubrir y compartir listas de reproducción, nueva música que encaja con los gustos del usuario. La *sidebar* de iLike que se integra con el reproductor de Apple iTunes también con el reproductor Windows Media Player escanea la biblioteca musical, recomienda nueva música, y permite al usuario conocer otras personas con gustos musicales similares.

2.4.4. FoxyTunes

FoxyTunes[57] es un *mashup* o aplicación híbrida ya que en esta aplicación se muestra contenido alojado en otras aplicaciones como son: Last.fm, LyrickWiki.org, Flickr.com, HypeMachine, Amazon, YouTube y Google.

El acceso a esta información se realiza a través de la web buscando la información que nos interese o bien mediante la descarga de un *plugin*. Con este *plugin* para Internet Explorer o Firefox se controla lo que reproducimos en nuestros reproductores de música mientras navegas por la Web de este modo la aplicación aprende más acerca de nuestros gustos musicales.

El nuevo portal musical, FoxyTunes Planet, tiene una filosofía de integración y agrupa distintos sitios de contenido musical y diferentes servicios en un único lugar.

2.5. Resumen

Como se ha visto hasta ahora, existen multitud de fuentes de información de contenido musical accesible a través de Internet, y también se han desarrollado distintas aplicaciones que gestionan contenido musical. En el sitio web programmableweb[58] se encuentran recogidas las anteriores aplicaciones y muchas otras. También quedan recogidos muchos *mashups* ya desarrollados.

Aún dada la gran cantidad de *mashups* de contenido musical ya desarrollados, la aplicación que se desarrolla con este trabajo aportará algo nuevo al menos al comienzo de su desarrollo. Es imposible conocer la existencia de otros desarrollos similares debido a la rápida y variada aparición de los mismos.

La aplicación que se desarrolla se puede vista como la unión de dos de las aplicaciones anteriormente expuestas, MusicMobs y FoxyTunes, considerando que la primera de ellas ya ha desaparecido.

Una unión porque como en MusicMobs se desarrollará una aplicación alojada en el ordenador del cliente que será la encargada de mantener sincronizada la información musical alojada en el ordenador del cliente y la información alojada en la aplicación web. Por otro lado, observando la aplicación web como un *mashup*, se accederán a los mismos servicios a los que ya accede la aplicación FoxyTunes, y así se complementará la información proporcionada por la aplicación de escritorio con la información proporcionada por las interfaces que permiten el acceso al contenido de los diferentes sitios web comentados. También se añade información de enlaces a ficheros *torrent* para descarga de contenidos, aspecto que no trata la aplicación FoxyTunes.

[56] iLike - http://www.ilike.com
[57] FoxyTunes - http://www.foxytunes.com
[58] programmableweb - http://www.programmableweb.com/

Ha diferencia de FoxyTunes la aplicación que se presenta en este proyecto está orientada más hacia lo que son las redes sociales, y a formar una comunidad, y no meramente a un portal de búsqueda de información como es FoxyTunes. Aunque la extensión hacia una red social no se ha desarrollado por falta de tiempo y dad la mayor complejidad que requiere.

CAPÍTULO 3

DISEÑO

En este capítulo se presenta el diseño general de la aplicación. Como ya se ha comentado, la aplicación final se divide en dos aplicaciones: una aplicación alojada en el ordenador del usuario (Aplicación de Escritorio) y una aplicación web, que representa un *mashup* (Aplicación Híbrida[1]).

El desarrollo del *mashup* se realizó de modo incremental incorporando el acceso a los diferentes servicios y contenidos, después de completar la fase que permitía almacenar la información musical proporcionada por los usuarios y de diseñar el sistema de gestión de usuarios. Es decir, se comenzó con un diseño básico de la aplicación que incluía simplemente el almacenamiento de la información proporcionada por los usuarios y a partir de ahí se fueron añadiendo funcionalidades siguiendo las directrices del desarrollo web ágil.

Una de estas necesidades fue la implementación de un sistema de caché para mejorar la usabilidad y potencia de la aplicación y que sera explicado más adelante.

Resumiendo, en este capítulo se explica qué estilo de *mashup* se ha decidido implementar para el desarrollo de la aplicación, de entre los dos estilos de *mashup* presentados en el capítulo 2: *mashup* del lado de servidor y *mashup* del lado de cliente.

También se explica por qué se ha decidido desarrollar una aplicación orientada a REST (*Restful application*) a partir de la información introductoria a REST presentada en el apartado 2.1.2, y cómo facilita este diseño el envío de la información musical del usuario al *mashup*.

Se explica la estructura de datos para almacenar la información proporcionada por el usuario, y cómo con el diseño del sistema de caché es necesario ampliar esta estructura con el objetivo de almacenar el contenido obtenido de los diferentes proveedores de información. Se comenta también cómo se realiza el acceso a los diferentes proveedores de servicios junto con la estructura que presenta dicha información para explicar el porqué de la estructura de datos diseñada.

También se detalla el diseño del sistema de gestión de usuarios permitiendo la creación de nuevos usuarios, la activación de los mismos mediante correo electrónico, etcétera.

A continuación, y en primer lugar, se mostrará un esquema simplificado de la arquitectura del sistema completo. Seguidamente se detalla la información musical de los usuarios que será enviada a la aplicación y que constituye la información fundamental que tomará como base la aplicación. En los puntos siguientes se detalla el diseño tanto de la aplicación de escritorio como de la aplicación web que depende del diseño de la parte fundamental de la aplicación, es decir, la estructura que seguirá la información musical que los usuarios proporcionarán.

[1] Se utilizará la voz inglesa *mashup*, en lugar del término aplicación híbrida, dada su gran aceptación.

3.1. Arquitectura del sistema completo

La figura 3.1 muestra un esquema general del sistema completo que se va a diseñar.

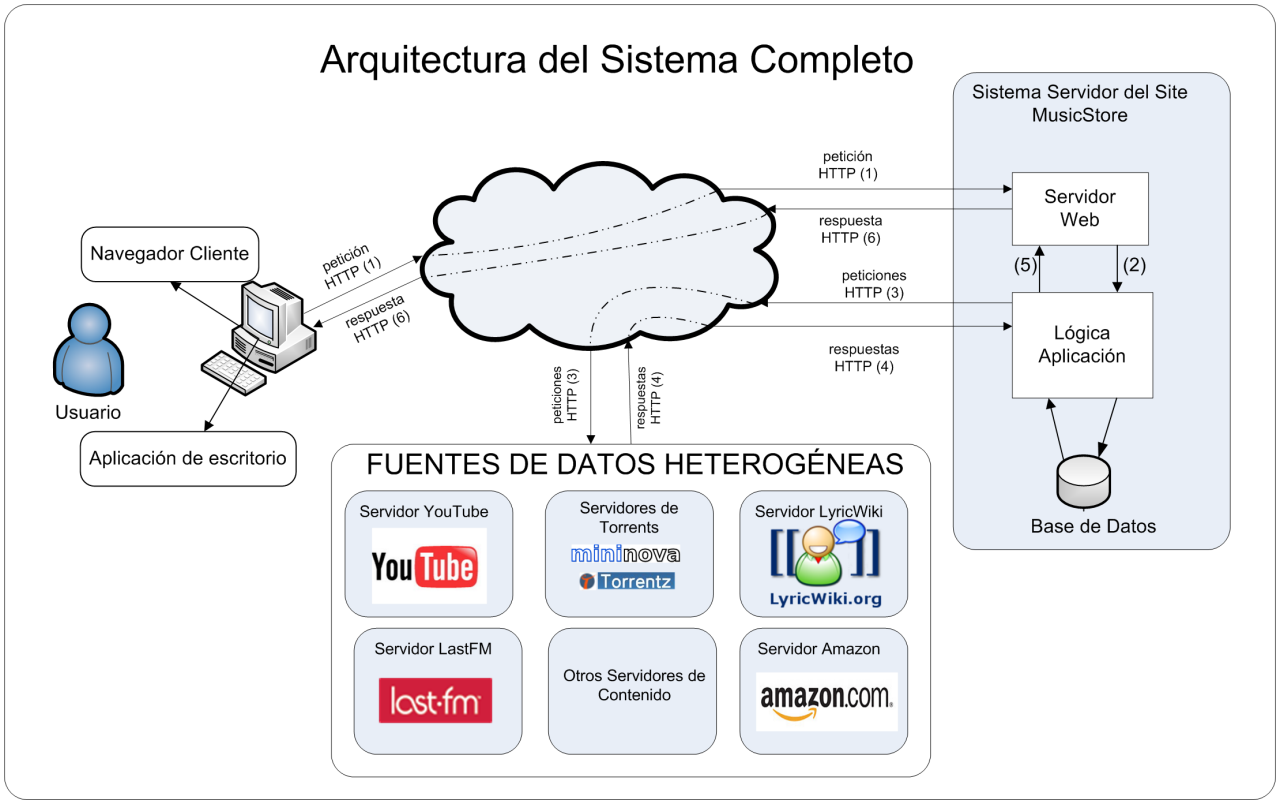

Figura 3.1: Esquema arquitectura del sistema completo

Por un lado se encuentran los diferentes modos de acceso a la aplicación web que son: a través de un navegador web o bien a través de la aplicación de escritorio. El acceso a través de un navegador web permitará al usuario navegar a través de sus contenido musical y obtener información relevante proporcionada por las diferentes fuentes de información y que son integrados por la aplicación web de un modo transparente para el usuario.

El acceso a la aplicación web a través de la aplicación de escritorio es utilizado para enviar la información musical que tienen el usuario en su máquina, utilizando la interfaz REST. La aplicación de escritorio también permite la sincronización del contenido entre la máquina del usuario y su cuenta en la aplicación web. La aplicación de escritorio está constituida por un interfaz de usuario, por un conjunto de clases que se encargar de procesar la información musical del usuario y por otro conjunto de clases que se encargan de la comunicación y sincronización con la aplicación web.

La aplicación web esta formada por: un servidor web, la lógica de aplicación y una base de datos. La aplicación web sigue un paradigma MVC y su desarrollado esta orientado a REST lo que proporciona ciertas ventajas. Una de estas ventajas es la existencia de una interfaz para la manipular los recursos de información que contiene la aplicación web. Esta interfaz es la que utiliza la aplicación de escritorio. La aplicación web cuenta con diversas bibliotecas de código que serán las encargadas de recuperar el contenido que proporcionan otras aplicaciones web. Ese contenido recuperado se integrará para proporcionárselo al usuario de la aplicación de modo transparente.

También se sitúan en dicho esquema los proveedores de contenido que permitirán darle a la aplica-

ción web su carácter de *mashup*. Los proveedores de contenido consultados son los siguientes: LastFm, YouTube, Lyricwiki, Amazon, Mininova y Torrentz. Cada uno de estos proveedores proporciona diferente información de contenido musical: vídeos musicales, letras de canciones, productos musicales, descargas de archivos musicales, etcétera.

En los siguientes apartados se detalla el diseño de cada una de las partes del sistema.

3.2. Información musical de usuarios

En primer lugar, se diseñó el modelo de datos que representa la información que proporciona el usuario a la aplicación, y también datos básicos de la propia aplicación como son el nombre de la aplicación, versión y el lenguaje por defecto de la misma, así como la estructura para la gestión y autenticación de usuarios. La imagen 3.2 muestra el esquema de entidades obviando la información no relevante.

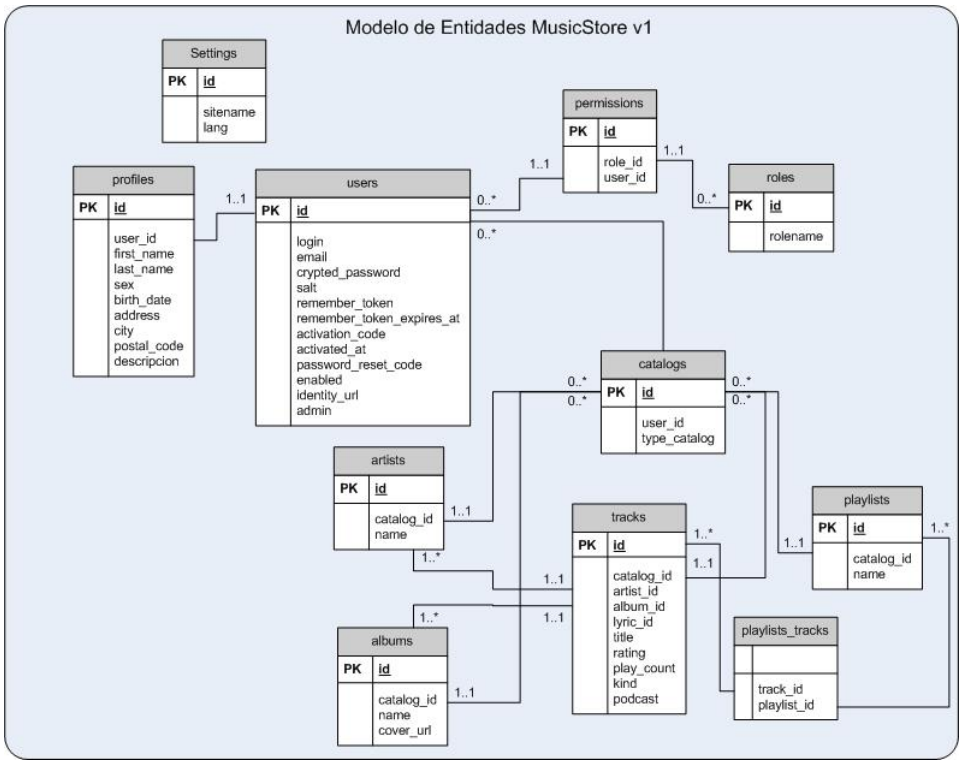

Figura 3.2: Diagrama de Entidades antes de implantar el sistema de caché

El esquema de entidades elegido surge como un punto de partida suficientemente abierto para poder realizar un desarrollo incremental, ya que una de las características de Rails es su capacidad para facilitar el desarrollo de aplicaciones incrementales. Esta primera versión es un prototipo y su estructura puede variar a lo largo del desarrollo de la aplicación. De hecho, el esquema anterior ya presenta variaciones respecto a la estructura inicial. En particular, en los modelos que permiten la gestión y autenticación de usuarios. La evolución de estos modelos se explicará en el apartado 3.3.1 donde se ha detallado todo lo relativo al sistema de gestión de usuarios.

De la información musical que proporcionan los usuarios destacan los siguientes aspectos:

- Un usuario puede disponer de varios `Catalogs`. Estos `Catalogs` o catálogos se corresponden con las diferentes fuentes de datos que nos pueda presentar el usuario, que en una primera apro-

ximación tomaremos como independientes, de ahí el atributo `type_catalog` que diferencia la fuente de datos específica.

- Un `Catalog` puede disponer de varios `Tracks` y `Playlists`. Los `Tracks` (canciones) son datos acerca de canciones que proporcione el usuario para ese catálogo y los `Playlists` (listas de reproducción) son agrupaciones de canciones que realiza el usuario.
- Un `Track` debe de tener al menos un `Album` y un `Artist` al que pertenece. También puede estar incluida en una o varias listas de reproducción. Aparte se incluye distinta información:
  - `title` título de la canción.
  - `rating` valoración del usuario para esa canción.
  - `podcast` indica si la canción se corresponde con un fichero de tipo podcast.
  - `kind` indica el tipo de fichero de la canción.
- Un `Album` puede estar formado por varias canciones, y varios artistas pueden haber contribuido a ese álbum. Como atributos se incluyen: `name` (nombre del álbum) y `cover_url` (URL donde se almacena una imagen de la carátula de dicho álbum, dato que es obtenido a través del contenido proporcionado por Last.fm[2]).
- Un `Artist` puede haber editado varios álbumes y por supuesto varias canciones. Como atributos se incluye: `name` (nombre del artista).

Por otro lado, también se cuenta con una entidad `Settings`, que se refiere a la configuración de la aplicación. Para esta entidad se han creado los atributos `sitename` y `lang`. El primero de ellos está relacionado con el nombre de la aplicación y su versión; el segundo se refiere al lenguaje por defecto de la aplicación. Como se ha comentado, Rails incluye soporte para internacionalización, que aunque no se ha utilizado en el proyecto, se podría incorporar en futuras versiones de manera sencilla.

3.3. Aplicación Web

El desarrollo de la aplicación web que compone el sistema de gestión de contenido musical se ha realizado de forma incremental siguiendo una metodología ágil. Por lo tanto, en este punto se irán presentando las decisiones de diseño que se fueron tomando de forma cronológica con el objetivo de ir mejorando las capacidades de la aplicación.

En primer lugar se debía decidir cuál sería la interfaz de la aplicación web para la comunicación con la aplicación de escritorio. Es decir, para recibir la información musical proporcionada por los usuarios de la aplicación. Como se explicó en 2.1.2 se plantean dos opciones fundamentales: la utilización de REST o la utilización de SOAP. Si bien la creación de este interfaz no es necesaria para el envio de información, será útil en el caso de que otras aplicaciones de terceros quieran acceder al contenido que aloja la aplicación web, como hace la aplicación al acceder al contenido de terceros a través de los interfaces que proporcionan.

Como ya se comentaba en ese mismo apartado, la utilización de REST es conveniente cuando sólo se quiere proporcionar servicios sencillos como son la consulta o envío de información. Para el caso de querer dar soporte de cifrado, envío fiable o seguridad sería necesario la utilización de SOAP. Sin embargo, en esta aplicación el interfaz será utilizado para el envío y consulta de información por lo que la implantación del paradigma REST es válido y facilitará enormemente el trabajo.

[2]se describe en la sección 4.4.1

Otro punto que apoya la utilización de REST es que a partir de la versión 1.2 de Rails, el propio *framework* cuenta con soporte para REST, lo que facilita enormemente el desarrollo de este tipo de aplicaciones. Construir la aplicación de este modo permite crear contenido, por ejemplo, desde una aplicación de escritorio (en este caso), y también que otras aplicaciones accedan a ese contenido de una manera sencilla, tal y como hace la aplicación en estudio al acceder al contenido de las otras aplicaciones. Por lo que se opta por lo que se denomina una *"RESTful Application"*, es decir, una aplicación con orientación REST para el diseño e implementación de la aplicación, al menos en la parte que incumbe a la información proporcionada por los usuarios acerca de la información musical que pretenden gestionar.

No se entrará en un detalle minucioso en cómo se construyen este tipo de aplicaciones en Rails, ya que no es el objetivo de este proyecto. Sin embargo, existe gran cantidad de información en Internet acerca de este tema. Para profundizar se recomienda la consulta de las siguientes referencias: [16], [19], [20], [21] y [22].

En cuanto a la cronología en el diseño de la aplicación web, en primer lugar se planteó la estructura REST de la aplicación que permitiera el almacenamiento de la información musical que iban a proporcionar los usuarios de la aplicación. En segundo lugar se estudió la realización del sistema de creación, autenticación y activación de usuarios.

Posteriormente, se fueron incluyendo las consultas a los diferentes servicios web de contenido musical para conformar el objetivo principal de la aplicación, que es complementar y aportar nueva información relevante a los usuarios. Por último, se estudió el modo de mejorar las consultas a los diferentes servicios web y el tratamiento de esta información con el objetivo de mejorar la experiencia de los usuarios. Para ello, se plantea un sistema de caché.

Todos estos puntos se irán ampliando con un mayor nivel de detalle en los apartados siguientes, y quedarán definidos completamente cuando se comente también su implementación. A continuación se detalla el diseño del sistema de gestión de usuarios.

3.3.1. Autenticación, Autorización y Registro de usuarios

A la hora de crear una aplicación web que se pretende sea utilizada por multitud de usuarios se debe establecer un modelo de seguridad específico. En esta aplicación nos decantamos por el Modelo de Seguridad AAA, de sus siglas en inglés *Authentication*, *Autorization* y *Accounting*. En castellano autenticación, autorización y registro de usuarios.

Las funciones que este sistema debe proporcionar en primer lugar son:

- *Login / Logout*. Operaciones que permiten el iniciar sesión y terminar sesión respectivamente en la aplicación web a usuarios previamente registrados.
- Registro de usuarios. Debe permitir la creación de nuevos usuarios.
- Manejo seguro de las contraseñas. Con lo que evitar la utilización fraudulenta de las mismas por parte de usuarios malintencionados, que puedan comprometer información de los usuarios.
- Métodos de autorización y control de acceso. Esto permitirá determinar qué privilegios tiene un usuario dentro de la aplicación web y a qué recursos tendrá acceso dentro de la misma.

Las funciones anteriores engloban las necesidades mínimas de todo sistema de gestión de usuarios. Todo este sistema apenas cambia de una aplicación a otra y suele ser bastante repetitivo y pesado de implementar. Sin embargo, gracias a la comunidad de desarrolladores de Rails, existen multitud de *plugins* para crear de forma segura y sencilla estos sistemas. Dado que la aplicación es orientada a REST, se utilizará el *plugin* Restful Authentication[3]. Este *plugin*, además de ser muy sencillo de instalar, permite

[3]Restful Authentication - http://github.com/technoweenie/restful-authentication

varias configuraciones diferentes.

Las funciones anteriores fueron implementadas en una primera etapa de diseño, como se comentará más adelante. Con esto se consiguió un diseño sencillo y sin complicaciones que satisfacía dichas necesidades. La estructura de datos para gestionar los usuarios que genera el *plugin* para estas funciones es la que se muestra en la figura 3.3[4]

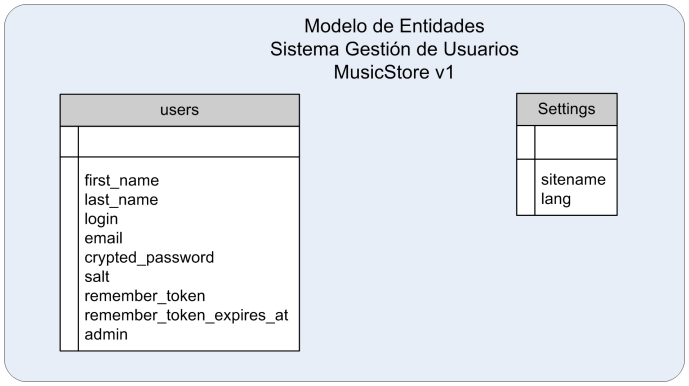

Figura 3.3: Diagrama de Entidades Gestión Usuarios v1

Sin embargo, con las funciones anteriormente comentadas puede que no se alcancen todas las funciones que debe tener una aplicación web social como la que se detalla en este proyecto, si se pretende que tenga éxito.

En una segunda iteración en el apartado de gestión de usuarios se decidió ampliar la funcionalidad de este sistema, basándose en el mismo *plugin* con una diferente configuración e incorporando nuevas bibliotecas de código y configuraciones. También se ampliaría la estructura de datos que utiliza dicho sistema.

De entre las distintas funciones que se añaden en esta segunda iteración al sistema se encuentran:

- Activación de cuentas por correo electrónico. Una vez el usuario crea su cuenta en la aplicación web se enviará un correo electrónico a la dirección proporcionada indicando el enlace cuya visita activará la cuenta del usuario, permitiéndole el uso de la aplicación. Con esto se evita la creación indiscriminada de usuarios por medio de robots. También fue considerada la utilización de un *captcha*[5] pero no se ha llevado a cabo finalmente, lo que protegería aún más la fase de registro de posibles robots. Estos *captcha* presentan una prueba de Turing para diferenciar máquinas de humanos que debe superarse para poder registrar satisfactoriamente un usuario.

- Aprobación o desactivación de cuentas por los administradores. El administrador de la aplicación web puede desactivar la cuenta de determinado usuario impidiendo el uso de la aplicación al mismo.

- Cambio y olvido de contraseñas. Estas funciones permiten cambiar la contraseña o crear una nueva si se olvido la anterior.

- Creación de roles, permisos y perfiles. Estos modelos van a permitir una mejor gestión de los usuarios en lo relativo al control de acceso a determinados recursos. Con los perfiles se separa la información de acceso a la aplicación del usuario de la información personal que el usuario proporcione,

[4]Las entidades aparecen en idioma inglés y no en castellano ya que Rails utiliza unos mecanismos de pluralización que pueden provocar la aparición de errores al utilizar un idioma diferente. También podrían aparecer incompatibilidades con algunos de los *plugins*.

[5]Más información en http://www.captcha.net/

permitiendo, a su vez, la modificación en caso necesario la información que compone el perfil de un usuarios sin comprometer otras entidades de la aplicación.

- Acceso por OpenID[6]. OpenID es una manera fácil de utilizar una única identidad digital a través de Internet, según se indica en su web oficial http://openid.net/. Con una identidad OpenID es posible iniciar sesión en muchos de los sitios webs más utilizados actualmente, con el mismo identificador y contraseña. Por lo tanto, el usuario no deberá recordar una nueva contraseña para utilizar la aplicación web. Este sistema de autenticación es utilizado actualmente en la mayor parte de las aplicaciones de lo que se considera web2.0.

Con esta última iteración en el diseño del sistema de gestión de usuarios, el sistema se completa además de una forma muy sencilla como se observará en el apartado de implementación del mismo. A la hora de la implementación no se detallará en profundidad su funcionamiento, pero sí la configuración utilizada, debido a que la implementación del *plugin* es de acceso público. La estructura de datos tras la ampliación de funciones se muestra en la figura 3.4

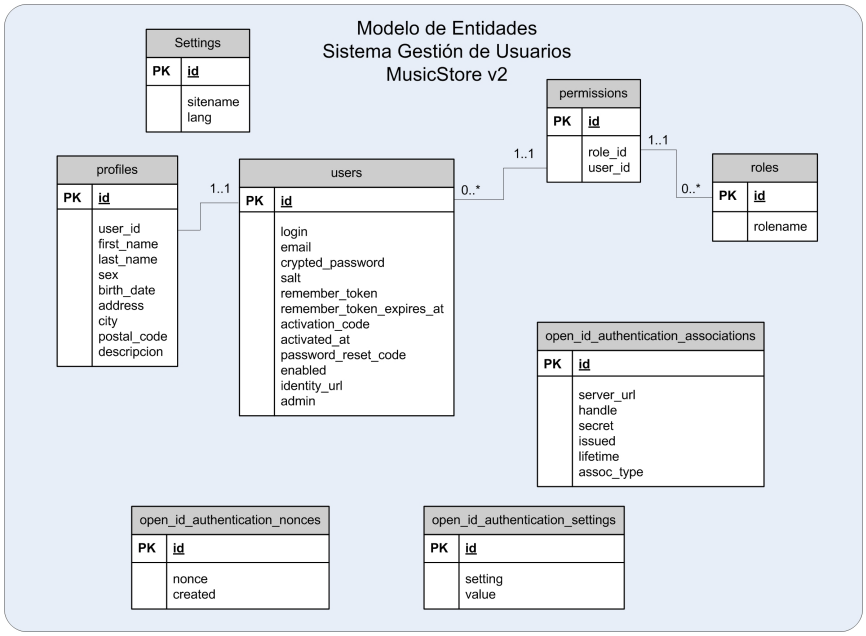

Figura 3.4: Diagrama de Entidades Gestión Usuarios v2

El hecho de ampliar la estructura de datos que soporta el sistema de gestión de usuarios con los modelos de datos roles y permisos es la de permitir un control más sencillo y centralizado del acceso de los usuarios a los diferentes recursos y espacios de la aplicación web. Estos modelos serán asignados a los diferentes usuarios a través de la interfaz web por los administradores de la misma.

En cuanto a la separación del modelo de usuario en los modelos usuarios y perfil (entidades *users* y *profiles*)nos sirve como ya hemos comentado anteriormente para separar la información de registro de la aplicación de la información personal que proporcione el usuario como puede ser: nombre, apellidos, fecha de nacimiento, etcétera. En el diseño inicial del sistema la información se mantenía en un único modelo usuario.

[6]Información sobre qué es OpenID - http://openid.net/what/

3.3.2. Gestión información proporcionada por los usuarios

En el apartado 3.2 se detalló la estructura de la información musical que proporcionarán los usuarios a través de la aplicación de escritorio. Pero lo que no se comentó fue cómo se traslada ese diseño a la estructura de la aplicación web.

La aplicación web, como ya se ha comentado, seguirá una orientación REST. Esto último, como ya se ha visto, quiere decir que se debe estructurar la información (recursos) mediante direcciones (URLs) que los identifiquen unívocamente.

Atendiendo a la estructura de datos vimos que un usuario puede disponer de varios `catalogs`. Esto podría ser modelado mediante la siguiente estructura de direcciones:

http://dominio/users/:user_id/catalogs
http://dominio/users/:user_id/catalogs/:id

En el primer caso para acceder a la colección de `catalogs` de usuario con identificador **:user_id**, y en el segundo para acceder al `catalog` con identificador **:id** de dicho usuario. Sin embargo, dado que va a ser necesario un nuevo subnivel (o anidamiento) para identificar todos los elementos que pertenecen a un `catalog`, se optó por el uso de las siguiente direcciones para identificar los `catalogs`:

http://dominio/catalogs
http://dominio/catalogs/:id

Durante la utilización de la aplicación web el usuario estará identificado desde el momento en el que inicia sesión en la aplicación, por lo que resultará sencillo controlar el acceso a los `catalogs`. Volviendo a la decisión anterior, en el desarrollo de aplicaciones REST se considera una mala práctica de diseño realizar un doble anidamiento que resultaría en direcciones del estilo:

http://dominio/users/:user_id/catalogs/:catalog_id/tracks
http://dominio/users/:user_id/catalogs/:catalog_id/tracks/:id
http://dominio/users/:user_id/catalogs/:catalog_id/artists
http://dominio/users/:user_id/catalogs/:catalog_id/artists/:id
http://dominio/users/:user_id/catalogs/:catalog_id/albums
http://dominio/users/:user_id/catalogs/:catalog_id/albums/:id

Siguiendo con la estructura de datos, ahora se deben enlazar los elementos (recursos) que pertenecen a un `catalog` específico. Se realizará mediante la siguiente estructura de direcciones:

http://dominio/catalogs/:catalog_id/tracks
http://dominio/catalogs/:catalog_id/tracks/:id http://dominio/catalogs/:catalog_id/artists
http://dominio/catalogs/:catalog_id/artists/:id http://dominio/catalogs/:catalog_id/albums
http://dominio/catalogs/:catalog_id/albums/:id http://dominio/catalogs/:catalog_id/playlists
http://dominio/catalogs/:catalog_id/playlists/:id

Con el anterior conjunto de direcciones es posible direccionar de forma unívoca cada uno de los recursos que componen la información musical proporcionada por los usuarios. Sin entrar en excesivos detalles en el apartado dedicado a la implementación, se explicará lo sencillo que es conseguir el anterior conjunto de direcciones, y que a partir de ahí sea el enrutador (*distpacher*) de la aplicación el que pase las peticiones que le llegan al controlador específico, y dentro de ese controlador a la acción determinada junto con todos los parámetros de la petición.

A continuación se explica la información que se mostrará al acceder a las diferentes direcciones para el formato de petición *.html*. Es decir, cuando se utiliza un navegador para acceder a la aplicación web, y sólo se pueden visualizar los recursos de información. El acceso para la creación de nuevos recursos, edición de los existentes, y la eliminación de los mismos no estará permitida salvo para los administradores de la aplicación. Los usuarios sólo podrán crear recursos a través de la aplicación de escritorio, peticiones en formato *.xml*.

También se comenta la estructura de las peticiones junto con la acción del controlador específico que será ejecutada.

Catalogs

El conjunto de peticiones de visualización de `catalogs` es:

Método	Path	Acción
GET	/catalogs	index
GET	/catalogs/:id	show

index En este caso si el usuario tiene varios `catalogs` se mostraría un listado de los mismos con enlaces a cada uno de ellos (acción *show*). Si el usuario solo cuenta con un `catalog` se le redirige directamente a la acción *show* de dicho `catalog`.

show En este caso se muestra toda la información relativa al `catalog` que se consulta. Se muestra un listado de álbumes, un listado de artistas y un listado de canciones que pertenecen a dicho `catalog`[7].

Tracks

El conjunto de peticiones de visualización de `tracks` es:

Método	Path	Acción
GET	/catalogs/:catalog_id/tracks	index
GET	/catalogs/:catalog_id/tracks/:id	show

index En este caso se muestra un listado de las canciones de dicho `catalog` junto con enlaces para la descripción de dicha canción, del álbum al que pertenece y al artista. También es posible realizar una búsqueda por el título de la canción.

show En este caso se muestra toda la información relativa al `track` que se consulta. Es decir, toda la información que ha proporcionado el usuario sobre esa canción.

Artists

El conjunto de peticiones de visualización de `artists` es:

Método	Path	Acción
GET	/catalogs/:catalog_id/artists	index
GET	/catalogs/:catalog_id/artists/:id	show

[7] En todos los listados de recursos se utiliza un sistema de paginación ya que los listados en algunos casos son enormes. En concreto se utiliza el *plugin* mislav-will_paginate, que se puede consultar en http://github.com/mislav/will_paginate

index En este caso se muestra un listado de los artistas de dicho `catalog` junto con un enlace para mostrar detalles del artista. También es posible realizar una búsqueda por el nombre del artista para acceder a su descripción (acción *show*).

show En este caso se muestra toda la información relativa al artista que se consulta.

Albums

El conjunto de peticiones de visualización de `albums` es:

Método	Path	Acción
GET	/catalogs/:catalog_id/albums	index
GET	/catalogs/:catalog_id/albums/:id	show

index En este caso se muestra un listado de los álbumes de dicho `catalog` junto con un enlace para mostrar detalles del álbum. También es posible realizar una búsqueda por el nombre del álbum para acceder a su descripción (acción *show*).

show En este caso se muestra toda la información relativa al álbum que se consulta.

Playlists

El conjunto de peticiones de visualización de `playlists` es

Método	Path	Acción
GET	/catalogs/:catalog_id/playlists	index
GET	/catalogs/:catalog_id/playlists/:id	show

index En este caso se muestra un listado de las listas de reproducción que pertenecen al `catalog`, junto con enlaces a cada una de ellas.

show En este caso se muestra toda la información relativa a la lista de reproducción que se consulta, junto con el listado de canciones que componen dicha lista de reproducción.

Con todo lo anterior se esclarece en parte el uso que se le puede dar a la aplicación como almacén de la información musical que además es accesible desde cualquier ordenador con conexión a Internet. Sin embargo, lo más interesante y el objetivo de esta aplicación es complementar dicho almacén con información que la propia aplicación puede obtener de otros sitios web o proveedores de contenido de forma ordenada e inteligente.

Es aquí donde entran en uso las diferentes APIs de consultas de información musical, las cuales serán detalladas en apartados siguientes. La información proporcionada a través de esas interfaces será mostrada en su mayoría dentro de las vistas/acciones *show*, es decir, esta información compone los detalles o información complementaria de la información que ha proporcionado el usuario. Las acciones *index* también incorporarán información complementaria, pero en este caso esta información será generada por la aplicación teniendo en cuenta la información obtenida de terceros y la información proporcionada por los usuarios.

Con el objetivo de aclarar la estructura anteriormente expuesta, se recomienda la lectura de la siguiente referencia [23], en dicha referencia se resume de forma breve la generación de las estructuras de rutas de una aplicación REST en Rails.

3.3.3. Discusión sobre la necesidad de un sistema de caché

Como se ha comentado en la introducción y a lo largo del texto, esta aplicación requiere la consulta de diferentes servicios web, que proporcionan contenido, disponibles a través de Internet con el objetivo de proporcionar información relevante al usuario de la aplicación a partir de la información que dicho usuario proporcione previamente.

Sin embargo, la consulta a diferentes servicios web presenta algunos problemas relacionados fundamentalmente con la usabilidad e interactividad y con el consumo de recursos de la aplicación. Cuando un usuario de la aplicación solicite contenido que debe proporcionar un tercero sufrirá cierto retardo hasta que obtenga el contenido solicitado. Se debe realizar una consulta al servidor externo de contenidos, procesar dicho contenido y mostrárselo al usuario. No parece lógico que si se realizan varias consultas para obtener el mismo contenido de un tercero se envíe una petición por cada consulta. Sería mejor plantear un modo de almacenar el contenido recibido en la primera consulta evitando realizar nuevas peticiones. Esto último permitirá reducir el número de consultas al proveedor de contenidos y reducir el consumo de recursos en términos de ancho de banda fundamentalmente. Los sistemas que permiten esto se denominan cachés.

Definición de caché . En informática se usan también, con este sentido, las expresiones *antememoria* o *memoria intermedia*. La caché o memoria intermedia es un conjunto de datos duplicados de otros datos originales con la propiedad de que los datos originales son costosos de acceder, normalmente en tiempo, respecto a la copia en el caché. Cuando se accede por primera vez a un dato, se hace una copia en el caché; los accesos siguientes se realizan a dicha copia, haciendo que el tiempo de acceso medio al dato sea menor. El conjunto de datos que puede almacenar una caché suele ser bastante reducido.

Este mecanismo puede utilizarse en diversos dispositivos/sistemas, como por ejemplo en una CPU, un disco duro o un navegador web.

En la definición anterior se relaciona el término caché con los navegadores web (caché web), y por lo tanto con uno de los clientes que pueden utilizar la aplicación web que estudiamos en este trabajo. La caché del navegador almacena copias de los documentos a los que ya se ha accedido, y como consecuencia posteriores peticiones pueden ser proporcionadas por la caché si se cumplen ciertas condiciones, disminuyendo considerablemente el tiempo de respuesta. El uso de esta caché en los navegadores reduce la utilización de ancho de banda, carga de servidores y retardo.

La implementación de un caché en los navegadores web responde al uso que hacen las personas de la web. Este uso presenta una alta *localidad temporal* y *localidad espacial*. Localidad temporal porque contenido que acaba de ser consultado es posible que vuelva a ser consultado con alta probabilidad en un corto intervalo de tiempo. Localidad espacial porque contenido que está siendo actualmente consultado puede enlazar con otro contenido que el usuario con alta probabilidad puede que consulte. En este último caso el navegador copiaría en caché el contenido al que hace referencia el contenido actual, para que cuando realmente acceda el usuario el retardo en el acceso sea mucho menor, ya que los datos ya están almacenados en caché.

Por lo tanto, una vez que la consulta a los diferentes servicios, la recuperación del contenido y su presentación al usuario estaba conseguido se planteó la necesidad de implementar un sistema de caché. Con los siguientes objetivos:

- **Reducción tráfico cursado.** Si el contenido que proporcionan los diferentes servicios ya está en cache y es contenido *«válido»*, no es necesario realizar nuevas peticiones HTTP, y el contenido es recuperado de la caché.

- **Reducción número de accesos a proveedores.** Algunos proveedores de contenidos limitan el número de peticiones a sus servicios.
- **Reducción tiempo de respuesta.** El contenido es proporcionado directamente por el servidor de la aplicación y no es necesario el acceso a terceros. Reduciendo el número de conexiones HTTP se reduce el tiempo de respuesta.
- **Personalización, adaptación y generación de contenido más sencilla.** Tener almacenado el contenido en el servidor facilita la personalización, adaptación y generación de nuevo contenido.

Aunque se explicará en detalle más adelante, el sistema de caché se integrará en el servidor de un modo transparente al usuario. Esta transparencia es sencilla de conseguir principalmente por el estilo de *mashup* elegido, ya que con este tipo de *mashup* es el servidor el que obtiene el contenido de terceros y no el navegador del cliente como proponen los *mashups* de lado de cliente. La Figura 3.5 incluye donde se situaría el sistema de caché utilizando un *mashup* de lado de servidor[8].

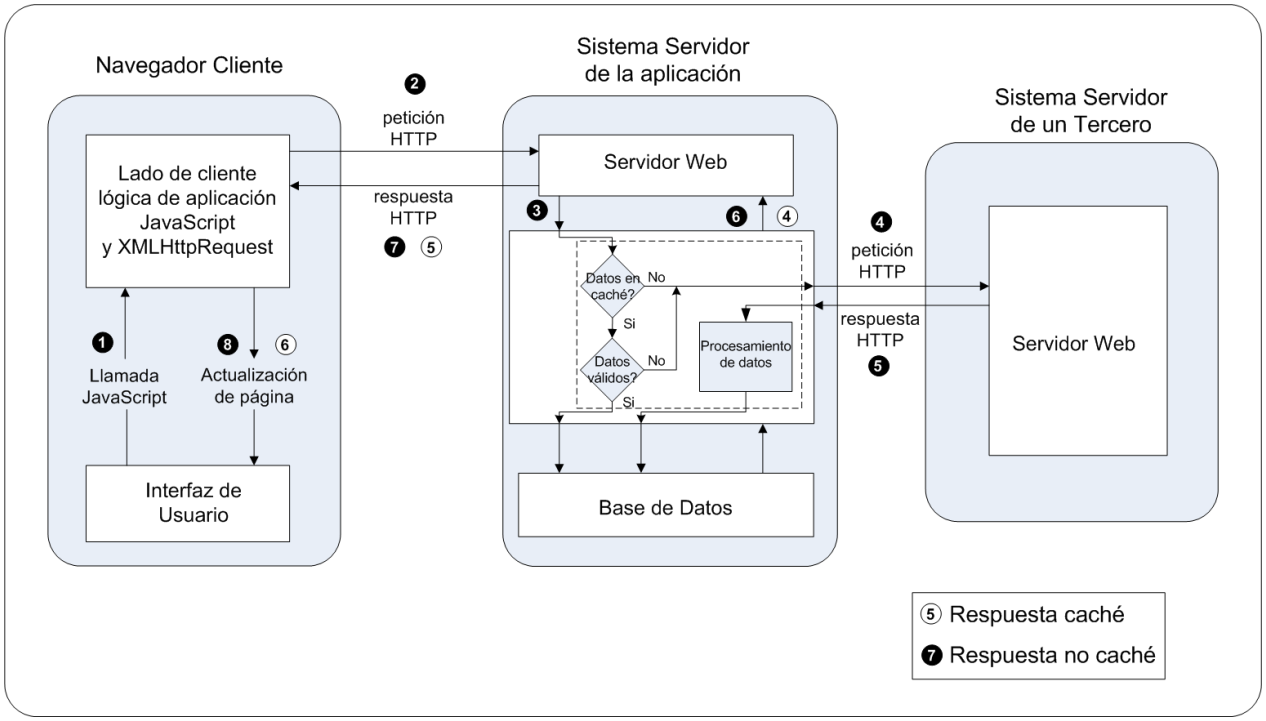

Figura 3.5: Integración de caché en un *mashup* de lado de servidor

En la siguiente sección se detallan diferentes opciones en el diseño y en el flujo del sistema de caché, ya que se debe alcanzar un compromiso entre los diferentes objetivos.

3.3.4. Diseño del Sistema de Caché

El sistema de caché va a ser implementado mediante la misma base de datos en la que se almacena la información musical que proporcionan los usuarios.

Este sistema de caché residirá en el servidor almacenando la información proporcionada por los proveedores de contenido. El modelo de *mashup* elegido para la aplicación permite incluir el sistema de caché en la aplicación de modo transparente para los usuarios, por lo que al usuario se le mostrará la

[8]La figura 3.5 se construyó en base a las figuras que aparecen en el artículo [2].

misma información, aunque verá reducido el tiempo de respuesta. En segundo término, y no menos importante, disponer de todo el contenido en la base de datos de la aplicación facilita la adaptación y/o creación de contenidos. Aunque este punto no se aborda en las primeras etapas de desarrollo de esta aplicación si se abordará posteriormente.

Como se ha comentado en la sección anterior, se van a evaluar diferentes soluciones a la hora de implementar un flujo de caché para la aplicación. Para cada una de ellas se analizarán sus ventajas y desventajas. Esto ayudará a tomar una decisión de diseño que será la que finalmente sea implementada.

Caché bajo demanda. En este caso los datos que nos proporcionan los proveedores de contenido a través de sus servicios se obtienen y almacenan en la base de datos cuando un usuario solicita por primera vez un determinado contenido a través de la aplicación. A partir de aquí, cuando un usuario (el anterior u otro distinto) acceda a ese mismo servicio, los datos provendrán directamente de esa caché (base de datos de la aplicación), por lo que no será necesario consultar de nuevo al proveedor de contenidos.

De este modo se **reduce el número de peticiones** al proveedor de contenidos y se **reduce el tráfico de red cursado**[9]. Sin embargo, el usuario que realice la primera consulta a un determinado servicio sufrirá **mayor retardo** respecto a servicios cuyo contenido ya se encuentre en caché. Esto se debe a que la aplicación deberá obtener dicho contenido de un proveedor, procesarlo y almacenarlo en la caché para, por último, mostrárselo al usuario. La inclusión de la caché implica un aumento del retardo respecto a la situación de partida, sin caché, ya que después de obtener el contenido de un tercero, y antes de mostrárselo al usuario, se incluyen operaciones para almacenar dichos datos en la caché.

Se podría sugerir que los servicios que pueda invocar un usuario en la página que está visitando podrían ser consultados de manera asíncrona por la propia aplicación. De este modo, si ha pasado el tiempo suficiente para que culminen esas peticiones asíncronas cuando el usuario realmente invoque el servicio los datos, se mostrarían rápidamente, dado que ya se encontrarían almacenados en caché. Esto no se planteará de momento en esta implementación, pero podría tenerse en cuenta de cara a posibles mejoras en usabilidad e interactividad de esta aplicación.

Caché y procesos en segundo plano. Otro punto de vista que resultará complementario para la obtención de contenido relevante para los usuarios de la aplicación es la ejecución de procesos en segundo plano. Estos procesos serán los encargados de acceder a los diferentes servicios en segundo plano, sin la intervención de ningún usuario, obtener el contenido que se proporciona a través de dichos servicios y almacenarlo en caché. Son procesos similares a las peticiones asíncronas/síncronas pero con un enfoque diferente, ya que aquí no hay ningún tipo de intervención indirecta/directa por parte de usuarios de la aplicación.

Por ejemplo: podría existir un proceso de fondo que realizara consultas al servicio de artistas similares de Last.fm. Este proceso realizaría una consulta a dicho servicio por cada uno de los diferentes nombres de artistas que haya proporcionado cualquiera de los usuarios de la aplicación. Almacenando todas las respuestas en el sistema de caché.

Este diseño implica que todos los servicios a los que accede la aplicación serán consultados al menos una vez. Esto conlleva un **número mucho mayor de consultas** que para el caso anteriormente descrito, ya que con seguridad habrá cierto contenido, y por lo tanto servicios, que no sean consultados nunca por ningún usuario. A pesar de que ciertos proveedores de contenidos presentan

[9] Relacionado con el esquema de *mashup* elegido

limitaciones en el número de consultas a los servicios que proporcionan, siempre será posible distribuir las consultas de los procesos de fondo a lo largo del tiempo.

Las ventajas de esta opción son claras. Por un lado, si suponemos que todos los procesos de fondo han sido ya ejecutados, es decir, se dispone de todo el contenido el usuario (el primero o el último) que solicite dicho contenido percibirá un **tiempo de respuesta a los servicios ofrecidos mucho menor**.

Otra ventaja muy importante de contar con toda esa información es que las tareas de personalización y adaptación de contenidos se pueden llevar a cabo de un modo más sencillo, ya que la aplicación cuenta con gran cantidad de información y con información muy relevante para los usuarios.

Se puede argumentar en contra de esta opción que cuando entre en funcionamiento no existirá ningún tipo de contenido. Pero una vez se ponga en marcha rápidamente el contenido ya estaría disponible en la caché y que cuando se vaya sumando información nueva por parte de nuevos usuarios con cada vez mayor probabilidad la información complementaria a la información del usuario ya estará almacenada en la caché. Aunque la parte de información que sea novedosa para la aplicación tendrá que ser recuperada por los distintos procesos de fondo.

Para el caso de que un nuevo usuario que acaba de enviar su información musical solicite seguidamente contenido relacionado con su información, y éste aún no esté en caché, existirá igualmente la posibilidad de solicitar dicho contenido bajo demanda, al menos en gran parte de los servicios, aunque no en todos ellos. Por lo tanto, ambas opciones son complementarias y pueden convivir, para unir sus ventajas y evitar sus inconvenientes.

Hasta ahora no hemos tenido en cuenta una característica del contenido que se proporciona en estos servicios y es que dicho contenido cambia, se corrige, se elimina, etcétera, por lo que será necesario que, cada cierto tiempo, se vuelvan a consultar los diferentes servicios y reemplazar el antiguo contenido con el nuevo. Para esto es interesante almacenar la fecha de adquisición de datos del proveedor de contenidos para cada uno de los servicios.

Se observa claramente que un parámetro muy importante a discutir es la cantidad de tiempo que debe transcurrir para que un contenido se considere *«no válido»* y se reemplace. Por supuesto, este parámetro podría ajustarse en función de la experiencia una vez la aplicación estuviera en marcha. Lo que es claro es que si la cantidad de tiempo es pequeña el número de peticiones a los servicios se dispararía, y si es muy grande puede que se este utilizando contenido no actualizado o erróneo. Antes de utilizar un contenido, se consultará en caché si dicho contenido se considera *«no válido»* o *«caducado»*. Se cree que el coste de esta consulta es menor que el coste de servir contenido no actualizado.

Al detallar las implicaciones de los procesos de fondo se comentó que puede darse el caso de que el contenido almacenado en caché nunca sea utilizado o consultado, o que lo sea en muy pocas ocasiones respecto a otro contenido. En estos casos sería interesante distinguir este contenido, los servicios que lo generaron, para que se considere como *«caducado»* cuando haya transcurrido un período de tiempo considerablemente superior, o incluso que esos datos se borren y no se vuelva a consultar dicho servicio. Este último extremo se podría llevar a cabo si el espacio disponible en la base de datos se reduce demasiado. Haciendo esto se evitará realizar una serie de peticiones, pudiendo aprovechar para realizar peticiones sobre contenido que consideramos que debe ser actualizado de forma periódica.

Para poder realizar esta diferenciación de contenidos se podría almacenar en la base de datos la fecha de la última consulta a esos datos. Un servicio con una fecha de consulta cercana en el tiempo implicaría un interés reciente por dicho contenido. Otra opción podría ser la introducción de un contador de acceso al servicio para evaluar la relevancia del mismo. Esta última opción no se introducirá en principio en

la aplicación. Este procedimiento de almacenar la fecha de consulta de los servicios implica un acceso más a la base de datos, por cada servicio consultado, ya que tenemos que actualizar dicha información. Sin embargo, puede que este acceso a base de datos se compense por el valor de tener los servicios más consultados actualizados. Esto requeriría un estudio con la aplicación en uso, para evaluar sus ventajas y sus inconvenientes, por lo que en un principio no será implementado.

A continuación se examina la estructura de la aplicación anterior a la ampliación de caché, y a qué partes afectará dicha implementación.

3.3.5. Ampliación estructura de datos para almacenar la caché

Antes de la implantación del sistema de caché la estructura de la base de datos de la aplicación era la que se muestra en la Figura 3.6[10]:

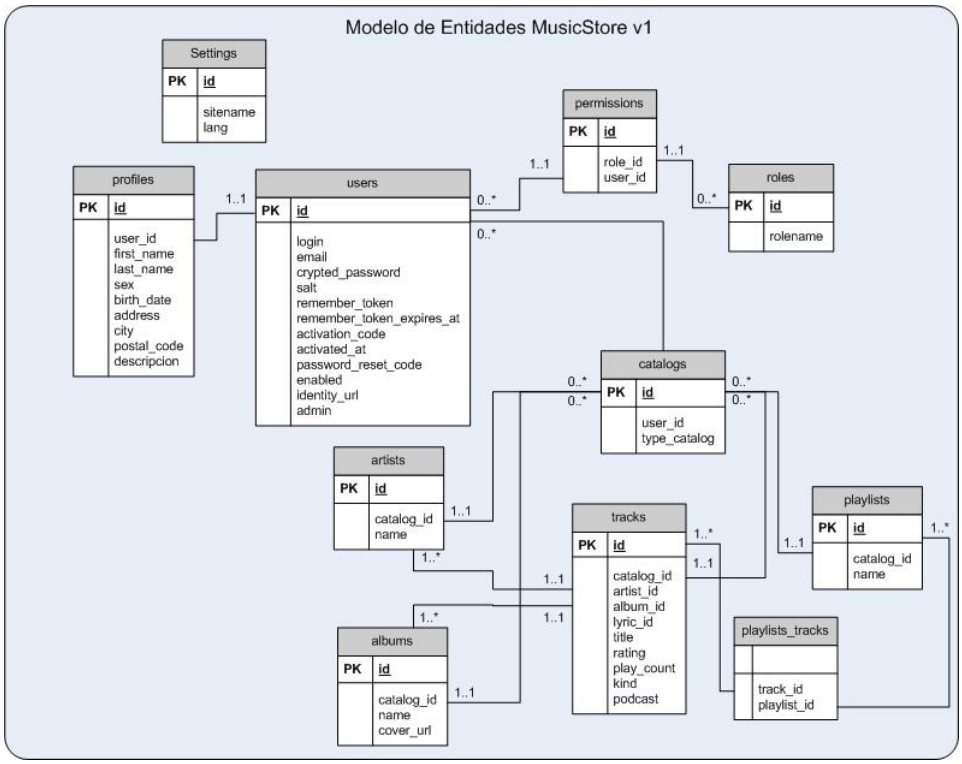

Figura 3.6: Diagrama de Entidades antes de implantar el sistema de caché

Dicha estructura permitía exclusivamente el almacenamiento de la información proporcionada por los usuarios e información del sistema de gestión de usuarios. Como ya se ha comentado, hasta este punto el contenido proporcionado por terceros era directamente enviado al cliente, sin ser almacenado en la base de datos. Con la ampliación propuesta se almacena dicho contenido.

Para conocer la estructura de la información que se almacena, a continuación se comenta a qué servicios accede la aplicación y cuales de ellos se almacenarán en caché.

Servicios consultados

Los siguientes servicios son los servicios que consulta la aplicación, junto con una explicación de los mismos y los parámetros necesarios para consultarlos:

[10]Figura idéntica a la presentada en la página 37 se utiliza de nuevo a modo de recordatorio

- **LastFM**. Con los siguientes servicios:
  - ***Similar Artists***. Recupera artistas similares a uno dado. Paramétros: nombre del artista.
  - ***Artist Top Albums***. Recupera los álbumes más relevantes de un artista dado. Paramétros: nombre del artista.
  - ***Album Info and Track List***. Recupera información y la lista de canciones de un álbum dado. Parámetros: nombre del artista y nombre del álbum.
  - ***Artist Top Tracks***. Recupera las canciones más relevantes de un artista dado. Parámetros: nombre del artista.
  - ***Artist Top Tags***. Recupera las etiquetas más relevantes asignadas a un artista dado. Parámetros: nombre del artista.
  - ***Similar Tracks***. Recupera canciones similares a una dada. Parámetros: nombre del artista y título de la canción.
  - ***Track Top Tags***. Recupera las etiquetas más relevantes asignadas a una canción dada. Parámetros: nombre del artista y título de la canción.
- **Amazon**. Catálogo de productos por el nombre del artista. Parámetros: nombre del artista.
- **Youtube**. Vídeos enlazados por el nombre del artista.
- **Torrentz**. *Feeds* de portales de archivos *.torrents* accedidos por el nombre del artista.
- **LyricWiki**. Letras de las canciones enlazadas por artista y título de la canción.

En principio la implementación del sistema de caché no cubrirá todos los servicios de los que la aplicación obtiene contenido, pero sí la mayoría. Los servicios para los que se implementará un sistema de caché son: los servicios proporcionados por *Last.fm*, por *Amazon* y por *LyricWiki*. En el caso de *Last.fm* los servicios de *tags* no serán almacenados en el sistema de caché. Además, en principio sólo los servicios de Last.fm podrán ser consultados bajo demanda.

Para poder llevar a cabo el sistema de caché se debe ampliar la estructura de datos para poder almacenar el contenido en la base de datos. A continuación se detalla el contenido que se recupera en las consultas a los diferentes servicios.

Contenido de los diferentes servicios

En este apartado se detallan ejemplos de peticiones y respuestas a los diferentes servicios. Aunque la aplicación no realizará directamente estas peticiones, ya que se dispone de bibliotecas que se encargan de realizar dichas peticiones y de estructurar el contenido con que responden los diferentes servicios. Aún así, es útil para entender la estructura de datos que se ha elegido para almacenar dicho contenido.

Servicios Last.fm

Muchos de los datos disponibles en la web de Last.fm están disponibles en varios formatos a través del API de servicios web de Audioscrobbler. Además, estos servicios web tienen una interfaz REST que utiliza XML y HTTP. Por lo tanto, la forma de obtener los datos es sencilla, es suficiente con realizar una petición *GET* HTTP a la dirección web (URL) del recurso (servicio) a consultar y si existe ese recurso nos responderán con el contenido correspondiente en formato XML. Se ha decidido utilizar la biblioteca de ruby **Scrobbler**[11], [24].

[11] Scrobbler http://www.railstips.org/2007/5/11/scrobbler-last-fm-for-ruby

Similar Artists Como ya se ha comentado, se necesita el nombre del artista sobre el que consultar sus artistas similares. La URI (Uniform Resource Identifier) a consultar es:

http://ws.audioscrobbler.com/1.0/artist/*nombre+artista***/similar.xml**

De aquí en adelante se utilizarán los ejemplos disponibles en la siguiente dirección web http://www.audioscrobbler.net/data/webservices/.

http://ws.audioscrobbler.com/1.0/artist/**metallica**/similar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<similarartists artist="Metallica" streamable="1"
        picture="http://userserve-ak.last.fm/serve/160/313628.jpg"
        mbid="65f4f0c5-ef9e-490c-aee3-909e7ae6b2ab">
<artist>
    <name>Pantera</name>
    <mbid>541f16f5-ad7a-428e-af89-9fa1b16d3c9c</mbid>
    <match>100</match>
    <url>http://www.last.fm/music/Pantera</url>
    <image_small>http://userserve-ak.last.fm/serve/50/262348.jpg</image_small>
    <image>http://userserve-ak.last.fm/serve/160/262348.jpg</image>
    <streamable>1</streamable>
</artist>
<artist>
    <name>Megadeth</name>
    <mbid>a9044915-8be3-4c7e-b11f-9e2d2ea0a91e</mbid>
    <match>94.77</match>
    <url>http://www.last.fm/music/Megadeth</url>
    <image_small>http://userserve-ak.last.fm/serve/50/163821.jpg</image_small>
    <image>http://userserve-ak.last.fm/serve/160/163821.jpg</image>
    <streamable>1</streamable>
</artist>
<artist>
     ...
</artist>
     ...
</similarartists>
```

Se observa que la información relevante acerca de un artista es la siguiente:

- *name*. Nombre del artista
- *mbid*. MusicBrainz Identifier. Identificador de la base de datos de MusicBrainz.
- *url*. URL del artista en Lastfm.
- *image_small*. URL a una imagen del artista.
- *image*. URL a una imagen del artista.

En cuanto a la relación de similitud entre el artista consultado (Metallica) y el contenido tenemos el atributo *match*. Cuanto mayor sean el valor de este atributo más parecido habrá entre los artistas.

Artist TopAlbums Se necesita el nombre del artista sobre el que consultar sus álbumes más relevantes. El formato de consulta sería:

http://ws.audioscrobbler.com/1.0/artist/*nombre+artista***/topalbums.xml**

http://ws.audioscrobbler.com/1.0/artist/**metallica**/topalbums.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<topalbums artist="Metallica">
<album>
    <name>Master of Puppets</name>
    <mbid>fed37cfc-2a6d-4569-9ac0-501a7c7598eb</mbid>
    <reach>75847</reach>
    <url>http://www.last.fm/music/Metallica/Master+of+Puppets</url>
    <image>
        <large>http://panther1.last.fm/coverart/130x130/1411810.jpg</large>
        <medium>http://panther1.last.fm/coverart/130x130/1411810.jpg</medium>
        <small>http://panther1.last.fm/coverart/50x50/1411810.jpg</small>
    </image>
</album>
<album>
    <name>Metallica</name>
    <mbid>3750d9e2-59f5-471d-8916-463433069bd1</mbid>
    <reach>66041</reach>
    <url>http://www.last.fm/music/Metallica/Metallica</url>
    <image>
        <large>http://cdn.last.fm/coverart/130x130/1411800.jpg</large>
        <medium>http://cdn.last.fm/coverart/130x130/1411800.jpg</medium>
        <small>http://cdn.last.fm/coverart/50x50/1411800.jpg</small>
    </image>
</album>
...
</topalbums>
```

Se observa que la información relevante a cerca de un álbum es la siguiente:

- *name*. Nombre del album
- *mbid*. MusicBrainz Identifier. Identificador de la base de datos de MusicBrainz.
- *reach*. Relevancia del álbum en Last.fm.
- *image/large*. URL a una imagen del álbum.
- *image/medium*. URL a una imagen del álbum.
- *image/small*. URL a una imagen del álbum.

También se tiene en cuenta que el álbum pertenece a un artista, con el que se realiza la consulta.

Album Info Se necesita el nombre del álbum sobre el que consultar su información para recuperar la lista de canciones que lo componen. El formato de consulta sería:

http://ws.audioscrobbler.com/1.0/album/*nombre+album***/info.xml**

http://ws.audioscrobbler.com/1.0/album/**Metallica/Ride %20the %20Lightning**/info.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<album artist="Metallica" title="Ride␣the␣Lightning">
    <reach>200631</reach>
    <url>http://www.last.fm/music/Metallica/Ride+the+Lightning</url>
    <releasedate>    0 1984, 00:00</releasedate>
    <coverart>
        <small>http://cdn.last.fm/coverart/130x130/1411812.jpg</small>
        <medium>http://cdn.last.fm/coverart/130x130/1411812.jpg</medium>
        <large>http://cdn.last.fm/coverart/130x130/1411812.jpg</large>
    </coverart>
    <mbid>456efd39-f0dc-4b4d-87c7-82bbc562d8f3</mbid>
```

```
    <tracks>
        <track title="Fight␣Fire␣With␣Fire">
            <reach>88400</reach>
            <url>http://www.last.fm/music/Metallica/_/Fight+Fire+With+Fire</url>
        </track>
        <track title="Ride␣the␣Lightning">
            <reach>98467</reach>
            <url>http://www.last.fm/music/Metallica/_/Ride+the+Lightning</url>
        </track>
        ...
    </tracks>
</album>
```

Se observa que la información relevante acerca de un álbum es la siguiente:

- *reach*. Relevancia del álbum en Last.fm.
- *url*. URL del álbum en Last.fm.
- *releasedate*. Fecha en la que se puso a la venta el álbum.
- *coverart/large*. URL a una imagen del álbum.
- *coverart/medium*. URL a una imagen del álbum.
- *coverart/small*. URL a una imagen del álbum.
- *mbid*. MusicBrainz Identifier. Identificador de la base de datos de MusicBrainz.

Como vemos, se amplia la información acerca del álbum respecto al servicio anterior con los siguientes datos: *releasedate* y *mbid*. A su vez, también se recupera la lista de canciones que componen el álbum, información que también debe ser almacenada en caché.

La información acerca de una canción (*track*) es:

- *title*. Título de la canción.
- *reach*. Relevancia de la canción en Last.fm.
- *url*. URL de la canción en Last.fm.

También se debe tener en cuenta que esta canción pertenece a un álbum y a un artista, los que se utilizaron para realizar la consulta.

Artist Top Tracks Se necesita el nombre del artista sobre el que consultar para obtener sus canciones más relevantes. El formato de consulta sería:

http://ws.audioscrobbler.com/1.0/artist/*nombre+artista***/toptracks.xml**

http://ws.audioscrobbler.com/1.0/artist/**Metallica**/toptracks.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mostknowntracks artist="Metallica">
<track>
    <name>Nothing Else Matters</name>
    <mbid></mbid>
    <reach>69411</reach>
    <url>http://www.last.fm/music/Metallica/_/Nothing+Else+Matters</url>
</track>
<track>
```

```
    <name>Enter Sandman</name>
    <mbid></mbid>
    <reach>67918</reach>
    <url>http://www.last.fm/music/Metallica/_/Enter+Sandman</url>
</track>
...
</mostknowntracks>
```

La información a cerca de una canción (*track*) es:

- *name*. Título de la canción.
- *mbid*. MusicBrainz Identifier. Identificador de la base de datos de MusicBrainz.
- *reach*. Relevancia de la canción en Last.fm.
- *url*. URL de la canción en Last.fm.

También se debe tener en cuenta que esta canción pertenece a un artista, el que se utilizó para realizar la consulta. Aquí debemos señalar que no se conoce el álbum al que pertenece la canción, por lo que no va a ser obligatorio conocer el álbum para almacenar una canción en la caché.

Similar Tracks Se necesita el nombre del artista y el título de la canción sobre la que vamos a consultar para obtener canciones similares. El formato de consulta sería:

http://ws.audioscrobbler.com/1.0/track/*nombre+artista/nombre+cancion***/similar.xml**

http://ws.audioscrobbler.com/1.0/track/**Metallica/Enter%20Sandman**/similar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<similartracks>
<track>
    <artist>
        <name>Iron Maiden</name>
        <url>http://www.last.fm/music/Iron+Maiden</url>
    </artist>
    <name>Run to the Hills</name>
    <match>24.84</match>
    <url>http://www.last.fm/music/Iron+Maiden/_/Run+to+the+Hills</url>
    <streamable>1</streamable>
</track>
<track>
    <artist>
        <name>Iron Maiden</name>
        <url>http://www.last.fm/music/Iron+Maiden</url>
    </artist>
    <name>The Number of the Beast</name>
    <match>22.84</match>
    <url>http://www.last.fm/music/Iron+Maiden/_/The+Number+of+the+Beast</url>
    <streamable>1</streamable>
</track>
...
</similartracks>
```

La información a cerca de una canción (*track*) y de su artista es:

- *artist/name*. Nombre del artista
- *artist/url*. URL del artista en Last.fm.
- *name*. Título de la canción.

- *url*. URL de la canción en Last.fm.

Como en el caso anterior, ***artist top tracks***, no se conoce el álbum al que pertenece la canción que es similar a la consultada. Como ocurría con el caso de similitud entre artistas el atributo *match* marca la similitud entre las canciones.

Cabe señalar que este servicio esta disponible en el API de AudioScrobbler sin embargo no es un servicio soportado por la biblioteca Scrobbler

Servicios Lyricwiki

La letra de las canciones será obtenida del portal LyricWiki. Este sitio web tiene un servicio web que nos permite recuperar esta información a través de SOAP ([7]), también disponen de una versión REST pero recomiendan utilizar el servicio SOAP.

En principio de este API se utilizan exclusivamente los siguientes métodos (servicios):

- **checkSongExists(artist, song)**. Este método indica si existe o no la canción del artista especificado en LyricWiki.
- **getSong(artist, song)**. Este método recupera la información asociada a la canción del artista especificado, dicha información incluye la letra de la canción.

A continuación se muestran ejemplos de las peticiones y respuestas que se obtienen en cada uno de estos servicios. Al utilizar SOAP las peticiones y respuestas siguen una estructura de información determinada.

Check Song Exists Como se vio anteriormente este servicio necesita como parámetros el artista y el título de la canción. A continuación se muestra un ejemplo del contenido de la petición y la respuesta que se obtiene[12].

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENC="http://schemas
    .xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
                <m:checkSongExists xmlns:m="urn:LyricWiki"
                SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
                        <artist xsi:type="xsd:string">Metallica</artist>
                        <song xsi:type="xsd:string">Nothing Else Matters</song>
                </m:checkSongExists>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV="http:
    //schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:si="http://soapinterop.org/xsd">
        <SOAP-ENV:Body>
                <ns1:checkSongExistsResponse xmlns:ns1="urn:LyricWiki">
                        <return xsi:type="xsd:boolean">1</return>
                </ns1:checkSongExistsResponse>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

[12]En las peticiones SOAP es necesario que este presente la cabecera *SOAPAction* con el nombre de la acción que quiere ser consultada

Como se observa en la respuesta este servicio indica si existe o no la canción consultada mediante un valor «*booleano*» dentro de la etiqueta `return`.

Get Song Con este servicio se obtiene la letra de la canción, a partir de un artista y del título de una canción. Igualmente presentamos un ejemplo de petición y respuesta.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENC="http://schemas
    .xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
                <m:getSong xmlns:m="urn:LyricWiki"
                SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
                        <artist xsi:type="xsd:string">Metallica</artist>
                        <song xsi:type="xsd:string">Nothing Else Matters</song>
                </m:getSong>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV="http:
    //schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:si="http://soapinterop.org/xsd" xmlns:tns="urn:LyricWiki">
        <SOAP-ENV:Body>
                <ns1:getSongResponse xmlns:ns1="urn:LyricWiki">
                        <return xsi:type="tns:LyricsResult">
                                <artist xsi:type="xsd:string">Metallica</artist>
                                <song xsi:type="xsd:string">Nothing Else Matters</song>
                                <lyrics xsi:type="xsd:string">So close no matter how far
Couldn&amp;apos;t be much more from the heart
Forever trusting who we are
And nothing else matters

Never opened myself this way
Life is ours, we live it our way
All these words I don&amp;apos;t just say
And nothing else matters

...

So close no matter how far
Couldn&amp;apos;t be much more from the heart
Forever trusting who we are
No, nothing else matters</lyrics>
                                <url xsi:type="xsd:string">http://lyricwiki.org/
                                    Metallica:Nothing_Else_Matters</url>
                        </return>
                </ns1:getSongResponse>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Como se observa en el contenido enviado como respuesta a la petición, se consigue la siguiente información sobre la canción:

- *lyrics*. Letra de la canción
- *url*. URL de la canción en Lyricwiki.

Estos datos pertenecen a un canción de un artista específico.

Servicios Amazon

En cuanto a los servicios consumidos del API Amazon Web Services (AWS) en su versión 4, únicamente se realizan búsquedas de productos (*ItemSearch*). En este caso dada la complejidad y tamaño del contenido que devuelve el API, no se cree necesario mostrar dicho contenido aquí, pero si en su apéndice dedicado, en concreto en la página 117. El archivo descriptor del API (archivo *WSDL (Web Service Description Language)*; para saber más sobre WSDL consultar [25]) puede ser consultado en http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl

En esta aplicación se utiliza la gema **Ruby-AAWS**[13], que ayuda a procesar el contenido devuelto. Además, cabe destacar, que la propia biblioteca también implementa por defecto un sistema de caché de peticiones con un tiempo de caducidad de 24 horas.

Una vez se ha detallado el contenido proporcionado por los diferentes servicios consultados que serán almacenados en caché se procede a comentar la estructura de datos definida para implementar el sistema de caché.

3.3.6. Estructura de Datos para caché

Con toda la información proporcionada en las sección anterior, ya es posible crear la estructura de datos necesaria para almacenar todo ese contenido en la base de datos de una forma estructurada.

La Figura 3.7 muestra las entidades o modelos de datos que se ha decidido utilizar para almacenar toda esta información. En la misma figura también se puede observar de qué proveedor de contenidos se obtiene la información que contiene cada uno de los modelos.

También es necesario enlazar toda esta estructura con la estructura de datos anterior. La Figura 3.8 muestra el esquema completo de entidades de la aplicación.

Como se puede observar en el diagrama de entidades completo, la información proporcionada por los diferentes servicios, que es almacenada en caché, es información complementaria a la información que proporciona el usuario. Esto último, proporcionar información relevante al usuario, es de hecho uno de los objetivos de este trabajo. Se complementa directamente con las principales entidades de la aplicación: *artist* con *info_artist*, *track* con *info_track* y con *lyric* y *album* con *info_album*.

Con el sistema de caché lo que se consigue principalmente es mejorar la experiencia del usuario con la aplicación, ya que el contenido reside directamente en la aplicación. Otra de las ventajas, que no se aborda en este momento, es la generación de nuevo contenido, lo cual se facilita enormemente si se cuenta con esta información adicional en la aplicación.

3.3.7. Flujo de caché

En este apartado se entra en detalle en el flujo que sigue la aplicación para proporcionar contenido utilizando la implementación de caché.

Un elemento fundamental es la utilización de procesos en segundo plano o de fondo (*background*), que serán los encargados de obtener el contenido de terceros en función de la información proporcionada por los usuarios de la aplicación. Estos procesos de fondo pueden ser automatizados y organizados en el tiempo. Las funciones que invocan dichos procesos son las mismas que invoca el usuario de la aplicación cuando quiere acceder al contenido de terceros, en las situaciones que se han denominado caché bajo demanda, por lo que la implementación de ambas situaciones es prácticamente idéntica, la diferencia fundamental radica en cómo se inician.

[13]Gema Ruby-AAWS http://www.caliban.org/ruby/ruby-aws/

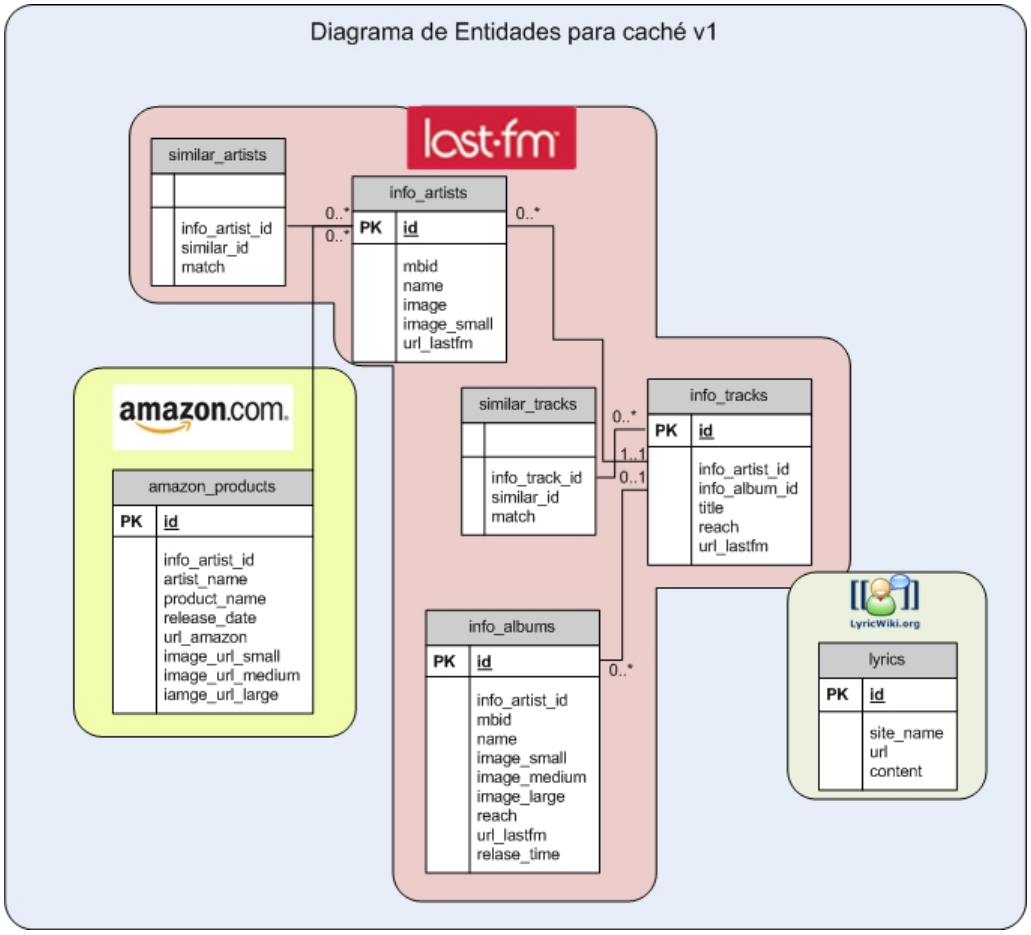

Figura 3.7: Entidades encargadas de almacenar la información de los servicios externos

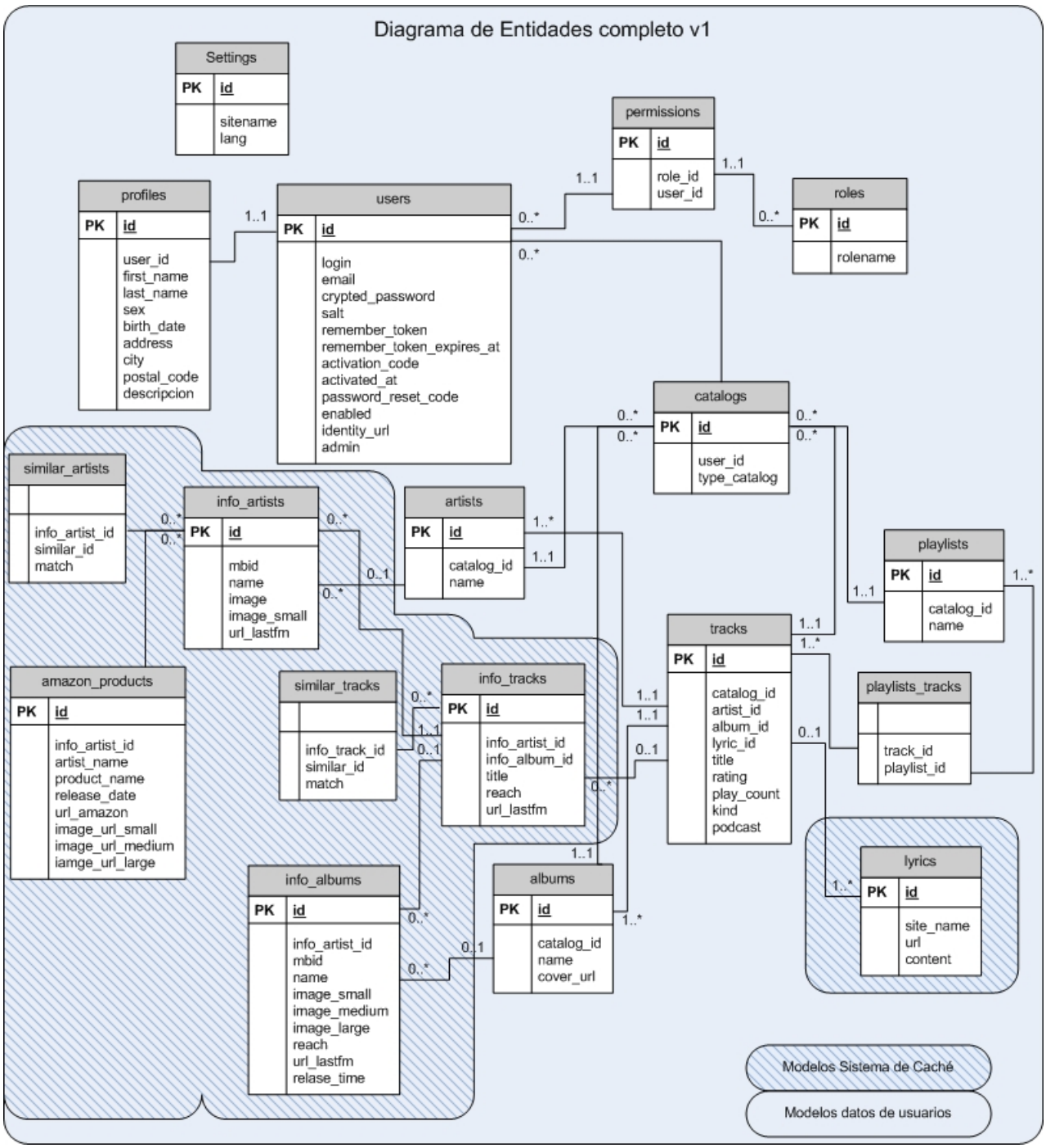

Figura 3.8: Diagrama de Entidades completo

Flujo de caché bajo demanda

A continuación se presenta una de las posibles situaciones en las que interviene nuestro sistema de caché. Este caso se dará para aquellos servicios que pueden invocarse directamente desde la aplicación web (contenido bajo demanda), como son los servicios de LastFM. Este escenario ya se presentó en la figura 3.5 (pág. 46), pero aquí se comentarán los pasos de dicho sistema.

1. ***Llamada JavaScrip***. Petición *JavaScript* realizada por el usuario sobre la interfaz web de la aplicación, solicitando contenido (solicitud bajo demanda).
2. ***Lógica de aplicación del lado del cliente***. El cliente web, el navegador, convierte la petición *JavaScript* en una petición HTTP hacia el servidor de la aplicación diseñada. Invocando una acción concreta de un controlador específico y con los parámetros oportunos.
3. ***Servidor Web***. El servidor web de la aplicación procesa dicha petición y la re-encamina al código de la aplicación invocando al controlador-acción específico. A partir de este punto es donde interviene el código que implementa el sistema de caché.
   a) ***¿Datos en caché?***. Se realiza una búsqueda del servicio solicitado en caché (en base de datos). Si los datos no están en caché se pasa al punto 4, sino se pasa al siguiente punto.
   b) ***¿Datos válidos?***. Comprobación de que los datos en caché son válidos. Si los datos están «caducados» se pasa al punto 4, si no se recuperan los datos de la caché y se devuelven.
4. ***Petición HTTP***. Se realiza una petición al proveedor de contenidos, sobre el contenido que solicitó el usuario, si los datos o bien no estaban en caché, o los datos de caché ya no son válidos.
5. ***Respuesta HTTP***. Se procesa la respuesta que nos proporciona el proveedor de contenidos y se almacena en caché, base de datos.
6. Si los datos estaban ya en caché o se han recuperado con éxito del proveedor de contenidos se genera la respuesta que se le pasa al servidor web.
7. El servidor web envía la respuesta a la petición del usuario con el contenido solicitado.
8. ***Actualización de página***. El navegador actualiza la página con los datos recuperados de la caché.

Como vemos, el sistema de caché es transparente para el usuario de la aplicación, ya que el usuario no sabe si los datos ya los tenemos almacenados en la caché o bien hemos consultado al proveedor de contenidos para obtener el contenido solicitado, aunque sí percibirá cierto retardo si los datos no estaban ya en la caché de la aplicación y deben ser recuperados del proveedor de contenidos y almacenados en caché.

Flujo de caché procesos de fondo

El anterior esquema mostraba los pasos para proporcionar contenido con el sistema de caché sobre servicios que puede solicitar el usuario de la aplicación, lo que se ha denominado contenido bajo demanda. Sin embrago, hay otros contenidos que serán consultados por la aplicación sin la intervención directa del usuario. Este tipo de contenidos se obtienen mediante procesos en segundo plano, que se ejecutarán de forma regular. Para este tipo de procesos el flujo de caché queda reducido al siguiente esquema gráfico (Figura 3.9[14]).

En los procesos de **fondo** los pasos que se realizan son los siguientes, salvo modificaciones muy específicas de algún servicio.

[14] La figura 3.9 se construyó en base a las figuras que aparecen en el artículo [2].

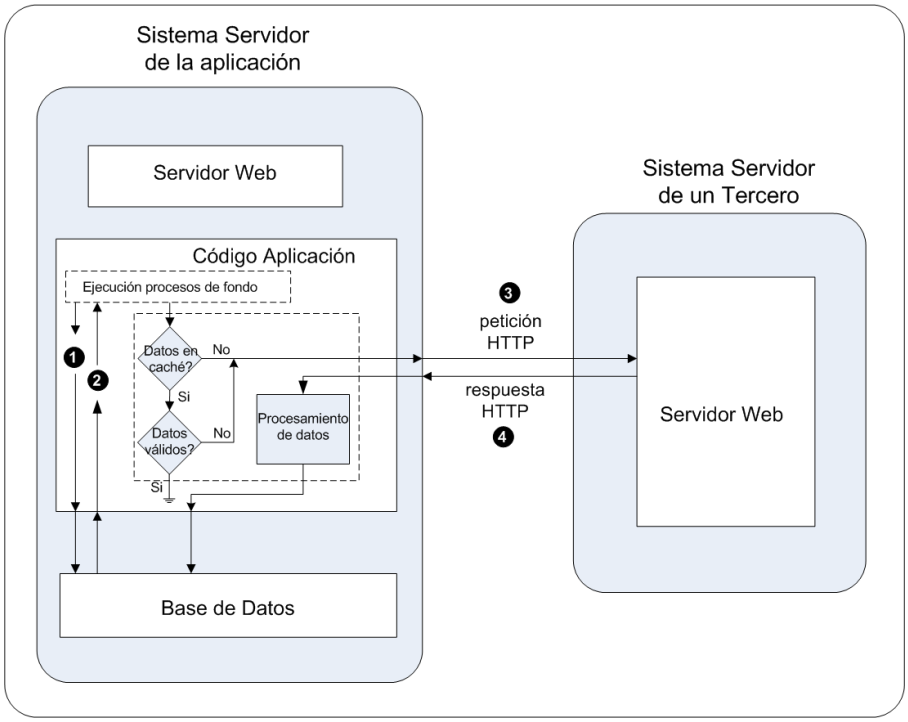

Figura 3.9: Ejecución de procesos de fondo para caché

1. Una vez se arranca un proceso de fondo de un determinado servicio, el proceso consultará la base de datos para obtener los parámetros sobre los que consultar el servicio indicado.

2. A partir de los parámetros recuperados el proceso iterará para realizar todas las consultas necesarias.

   a) ***¿Datos en caché?***. Se realiza una búsqueda del servicio solicitado en caché (en base de datos). Si los datos no están en caché se pasa al punto 3. Si no, se pasa al siguiente punto.

   b) ***¿Datos válidos?***. Comprobación de que los datos en caché son válidos. Si los datos están «caducados» se pasa al punto 3. Si no esta iteración termina.

3. ***Petición HTTP***. Se realiza una petición al proveedor de contenidos, sobre el contenido que se solicita, ya que los datos o bien no estaban en caché, o los datos de caché ya no son válidos.

4. ***Respuesta HTTP***. Se procesa la respuesta que nos proporciona el proveedor de contenidos y se almacena en caché, base de datos.

Como se observa, los procesos de fondo van a compartir su implementación con las consultas bajo demanda. Con este planteamiento los procesos de fondo se comportan como usuarios que consultan todas las combinaciones de un determinado servicio.

Sin embargo, los procesos de fondo pueden ser optimizados, aunque en este proyecto no se implementará por falta de tiempo. En el proceso anteriormente comentado se realiza una primera consulta para obtener todas las posibles opciones de consulta de un determinado servicio (el relacionado con el proceso de fondo que se ejecuta), y luego para cada una de las opciones se consulta si los datos de dicho servicio están en caché y si son válidos en el caso de que existan. Todo esto podría realizarse en la primera consulta, y que el proceso solo iterara sobre los parámetros del servicio cuyo contenido debe ser actualizado o adquirido.

3.4. Aplicación de Escritorio

Al haber diseñado una aplicación web orientada a REST se ha facilitado el trasladar los datos del usuario a la aplicación web. Para ello se utiliza una aplicación de escritorio desarrollada en Java, ya que es uno de los lenguajes más utilizados en los últimos años, es multiplataforma y está ampliamente instalado en máquinas de usuarios.

Como único requisito para el uso de la aplicación de escritorio, se requiere que el usuario tenga instalada una máquina virtual de Java, al menos la versión 6, ya que la aplicación será un ejecutable java. De este modo se consigue una aplicación multiplataforma con unos mínimos requisitos. Esta aplicación a su vez se puede dividir en tres partes: la interfaz de usuario, las bibliotecas de procesamiento de información musical y modelado de datos, y las bibliotecas de comunicación con la aplicación web.

La interfaz de usuario a sido desarrollada mediante *Swing*[15], biblioteca gráfica que forma parte del núcleo de Java. Esta interfaz consiste en una interfaz que guía al usuario a través de los pasos que debe seguir para enviar su información musical a la aplicación.

Por otro lado se encuentran todas las bibliotecas que se encargan de todo el procesamiento de la información musical que proporciona el usuario, su modelado y su adecuación para su posterior envío a la aplicación web, a través de su interfaz REST.

Se han planteado tres métodos de acceso al contenido musical del usuario: procesando la biblioteca del reproductor iTunes, la biblioteca del reproductor Amarok o bien realizando la búsqueda a través de las etiquetas (*tags*) de los ficheros de audio del usuario. La elección de estos tres métodos se basa en la gran utilización del reproductor iTunes (usuarios de Windows y de Mac) y del reproductor Amarok (usuarios de Linux), cubriendo así gran número de usuarios. El tercer método es un método independiente del tipo de reproductor que utilice el usuario. Se debe resaltar que la información musical de la que disponga el usuario se considera catalogada y etiquetada de forma correcta, sin entrar en cómo conseguir este propósito.

Se ha creído conveniente reducir la complejidad de toda la aplicación de escritorio ya que no se considera parte fundamental de este trabajo, aunque sí una parte que debe satisfacer unos mínimos requisitos.

A continuación explicaremos el diseño que se ha seguido en cada una de las partes que conforman la aplicación de escritorio.

3.4.1. Interfaz de usuario

Como ya se ha comentado anteriormente, la interfaz de usuario ha sido desarrollada mediante *Swing*. Básicamente, la interfaz guiará al usuario para que pueda enviar su información musical a la aplicación web, que la guardará en la base de datos.

Los pasos que guiarán al usuario y que debe ofrecer la interfaz de usuario son los siguientes:

1. **Autenticación de usuarios**. En primer lugar se debe autenticar al usuario que utiliza la aplicación de escritorio con la aplicación web. Se comprobará que el usuario dispone de una cuenta de usuario activada en la aplicación web. En caso de que no disponga de una cuenta activada, no podrá seguir con el uso de la aplicación de escritorio.

2. **Selección procedencia información musical**. En este segundo paso se solicita al usuario que indique de dónde se van a obtener su información musical, con las siguientes opciones:

[15] Ampliar información en http://java.sun.com/javase/technologies/desktop/

- **iTunes**. Se desplegará una ventana en la que el usuario deberá presentar el fichero de biblioteca que mantiene el reproductor iTunes.
- **files**. Se desplegará una ventana en la que el usuario deberá seleccionar el directorio sobre el que se realizará la búsqueda de su información musical.
- **amarok**. Se desplegará una ventana en la que el usuario deberá seleccionar el fichero de biblioteca que mantiene el reproductor Amarok.

3. **Procesamiento información musical**. Inmediatamente después de la selección del paso anterior la aplicación comenzará a obtener la información del usuario y a procesarla. Este procesamiento será explicado en mayor detalle en apartados posteriores. Una vez terminado el procesamiento se pasará al punto siguiente.

4. **Selección de datos de envío y sincronismo**. En este último paso, se mostrará al usuario en forma de lista las listas de reproducción (en el caso de iTunes y Amarok) o la biblioteca completa que componen la información musical aportada por el usuario, junto con la posibilidad de seleccionar dos opciones para el sincronismo de datos:

   - **Borrar y Cargar todo**. Con esta primera opción lo que se busca es realizar una nueva carga de datos, eliminando previamente toda la información musical que aportó el usuario.
   - **Sincronismo con el servidor**. Con esta segunda opción se pretende realizar una sincronización entre los datos que aporta el usuario y los datos que aportó el usuario en alguna ocasión anterior. Actualizando los datos, por ejemplo número de reproducciones, pero sin eliminar la información que ya no exista en la información aportada por el usuario.
   - En el caso de no seleccionar ninguna de las opciones anteriores, se enviarán los datos modificados respecto a envíos anteriores, siempre que existan ficheros de registro[16]. Si no existen ficheros de registro o es la primera vez que se envían datos a la aplicación se enviará toda la información seleccionada.

3.4.2. Procesado información musical del usuario

Una vez el usuario indique cómo obtener la información de su contenido musical, la aplicación de escritorio llevará a acabo el procesamiento oportuno para agrupar toda esa información. Con esa información se generarán los recursos de información que mediante REST serán enviados a la aplicación web de forma ordenada. Por supuesto, la estructura de los recursos de información que se generan y que son enviados a la aplicación web debe coincidir con la esperada por la misma.

Al proceder la información musical del usuario de tres fuentes diferentes, el procesamiento se realizara de tres modos, cada uno de ellos específico de una fuente. Todos los modos construirán recursos de información con la misma estructura y que serán los que comprenda la aplicación web.

Procesamiento iTunes

Centrándose en el caso del reproductor de iTunes, su biblioteca está constituida por un fichero XML. Este fichero XML es del tipo *PropertyList* y se encarga de almacenar la información de la biblioteca. Este tipo de fichero fue introducido por Apple en MacOS 10.0 junto con una DTD (Document Type Definition) pública[17]. Un ejemplo de este tipo de ficheros *property list* sería el siguiente.

[16] En el apartado dedicado a la comunicación con la aplicación web y al sincronismo

[17] Consultar Manual de plist en developer.apple.com para más información - http://developer.apple.com/documentation/Cocoa/Conceptual/PropertyLists/PropertyLists.html

```
      <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
      <dict>
      <key>Year Of Birth</key>
    <integer>1965</integer>
    <key>Pets Names</key>
    <array/>
    <key>Picture</key>
    <data>
      PEKBpYGlmYFCPA==
    </data>
    <key>City of Birth</key>
    <string>Springfield</string>
    <key>Name</key>
    <string>John Doe</string>
    <key>Kids Names</key>
    <array>
      <string>John</string>
      <string>Kyra</string>
    </array>
  </dict>
</plist>
```

Para el procesamiento de este tipo de ficheros XML se utilizará el API *Configuration* desarrollado en Java de Apache (*Jakarta Commons Configuration*[18]). Este API facilitará enormemente la extracción de la información para construir el formato común de información ya que el API cuenta con clases específicas para el *parseo* de ficheros *PropertyList*.

Un primer paso que se considera necesario al procesar este tipo de ficheros, dado el tamaño que pueden alcanzar, es eliminar la información que no va a ser utilizada en la aplicación web, como por ejemplo: *bit rate*, *sample rate*, *composer*, *size*, *total time*, *artwork count*, etcétera. Eliminando esta información se consigue reducir el tamaño del fichero de biblioteca considerablemente, lo que ayudará a reducir el consumo de memoria de la aplicación. La información útil es almacenada en un nuevo fichero que es el que servirá de base a partir de este momento. Este fichero tendrá el nombre *processed.xml*. En el apéndice C se muestra un fichero de biblioteca de iTunes original (pág. 122) y también se muestra el resultado de eliminar del mismo la información que no va a ser utilizada en la aplicación web (pág. 124).

Posteriormente se procede a ir procesando una a una cada una de las listas de reproducción que conforman el fichero. Dentro de cada lista de reproducción se procesarán todas las canciones que las componen. Con esa información se formará un objeto biblioteca (`Library`), el cual almacena toda la información que proporciona el usuario. Este objeto `Library` será el que compartan todos los modos de uso de la aplicación.

Procesamiento directo de ficheros de audio

En este caso la información del usuario debe ser obtenida de las etiquetas (*tags*) de los ficheros de audio que se encuentren dentro del directorio de búsqueda que indicó el usuario.

En el directorio indicado se realizará una búsqueda iterativa y, a partir de cada fichero de audio, se constituirá un objeto *Track*, el cual representa a una canción, y que también es un objeto común a los tres modos, a partir de la información que se encuentre en las etiquetas del fichero.

Para la obtención de dicha información se utilizarán hasta tres bibliotecas de código. Todas ellas son capaces de leer las etiquetas de ficheros MP3 (MPEG-1 Audio Layer 3), pero se ha comprobado que no son capaces de forma independiente de leer todos los ficheros MP3 que se han utilizado en la aplicación a

[18] Jakarta Commons Configuration - http://jakarta.apache.org/commons/configuration

modo de pruebas. Sin embargo, se utiliza una biblioteca como prioritaria. Estas tres bibliotecas de código son las siguientes:

- **MyID3**. Biblioteca de código para etiquetas ID3[19] escrita en Java por Charles M. Chen. Será nuestra biblioteca prioritaria por ser la que más tipos de etiquetas reconoce, y la que más actualizada se encuentra. Primero se intentará extraer la información del fichero con esta biblioteca y en caso de que no sea posible se intentará extraer con las dos siguientes bibliotecas. Se utilizará la versión 0.82 liberada el 22 de Julio de 2008. Para más información consultar http://www.fightingquaker.com/myid3/.
- **JID3**. Biblioteca escrita en Java y bajo licencia LGPL (GNU Lesser General Public License), que proporciona la funcionalidad necesaria para editar etiquetas ID3, que son las que se utilizan en los ficheros MP3. Se utiliza la versión 0.46 del 10 de Octubre de 2005. Para más información consultar http://jid3.blinkenlights.org/.
- **jid3lib**. Biblioteca igualmente escrita en Java. Se utilizará la versión 0.5.4 del 15 de Marzo de 2006. Para más información consultar http://javamusictag.sourceforge.net/.

Procesamiento Amarok

El reproductor multimedia Amarok se considera uno de los más utilizados por los usuarios de sistemas *UNIX* y *GNU-LINUX*, de ahí que se halla considerado como fuente de información musical de este grupo importante de usuarios.

Este reproductor almacena la información musical que le aporta el usuario en una base de datos. Esta base de datos en la opción por defecto es del tipo *sqlite*, aunque posibilita la utilización de otros gestores de bases de datos como son: Mysql, PostgreSQL, etcétera. Aunque no se ha comprobado por falta de tiempo, se considera que la estructura de la base de datos del reproductor es independiente del gestor de base de datos que se utiliza. También decir que solo se ha considerado la utilización de SQLite.

SQLite es una base de datos transaccional bajo la forma de un fichero que se mantiene en la memoria de las aplicaciones, sin necesidad de contar con un servidor y sin configuración. Está escrita en C y el código es de dominio público.

Son muchas las aplicaciones que utilizan SQLite, desde aplicaciones de escritorio, PDAs, teléfonos móviles y hasta reproductores MP3.

Para acceder a este tipo de base de datos desde una aplicación Java se ha utilizado la biblioteca SQLiteJDBC. SQLiteJDBC es un *driver* Java JDBC para SQLite, que permite acceder al contenido de bases de datos SQLite de forma muy sencilla. Se ha utilizado la versión 0.52 del *driver* con fecha 28 de Junio de 2008. Para obtener más información, consultar http://www.zentus.com/sqlitejdbc/.

Cuando se explique la implementación se comentará la estructura de datos que sigue el reproductor Amarok.

3.4.3. Comunicación con aplicación web y sincronismo

En esta sección se detallará el diseño elegido para que la aplicación de escritorio se comunique con la aplicación web, es decir, como será estructurado el envío de peticiones al servidor junto con la información que debe contener cada unas de estas peticiones.

Una vez el usuario indique cómo obtener la información de su contenido musical, la aplicación de escritorio llevará a acabo el procesamiento oportuno para agrupar toda esa información. La información recopilada se almacenará en un objeto `Library`, almacenado en memoria, como ya se comento

[19] ID3 es un contenedor de metadatos usado generalmente junto a ficheros de audio en formato MP3. Permite almacenar información relativa al fichero de audio en el propio fichero como puede ser: título, artista, álbum, número de canción, etcétera.

anteriormente. Con esa información se generarán los recursos de información que serán enviados a la aplicación web de forma ordenada. Posteriormente, el usuario ya estará en disposición de visualizar esta información en la aplicación y obtener información complementaria de los distintos proveedores de contenidos. Por supuesto, la estructura de los recursos de información que se generan y que son enviados a la aplicación web debe coincidir con la esperada por la misma.

En este mismo apartado se explicará que solución se ha diseñado para ofrecer cierta capacidad de sincronización entre la aplicación de escritorio, es decir los datos del usuario, y la información que ese mismo usuario ya tenga almacenada en la aplicación web.

Envío de información a la aplicación web

Como ya se vio en el apartado 3.2, el diagrama de entidades de la información que proporcionan los usuarios está compuesto fundamentalmente por 5 entidades: `Catalogs`, `Tracks`, `Albums`, `Artists` y `Playlists`. Atendiendo a lo que se ha aprendido a cerca del paradigma REST, es posible crear entidades de cualquiera de estos tipos de recurso enviando peticiones a las siguientes URLs[20]:

Método	Path	Acción
POST	/catalogs.xml	create
POST	/catalogs/:catalog_id/tracks.xml	create
POST	/catalogs/:catalog_id/artists.xml	create
POST	/catalogs/:catalog_id/albums.xml	create
POST	/catalogs/:catalog_id/playlists.xml	create

Con estas peticiones se pretende crear entidades de cada uno de los tipos de recursos. Pero cada tipo de recurso necesita cierta información, como se explicó en el apartado 3.2, y como se puede observar en la Figura 3.2. Por ello, es necesario enviar en el cuerpo de la petición esa información. Será información en formato XML, ya que es el formato indicado en la petición.

En cuanto al formato de la petición, se eligió el formato XML ya que nos permite estructurar la información de una manera clara e inteligible. Además Rails cuenta con soporte para procesar este tipo de estructuras de información, pudiendo transformar dicha información en un objeto de la clase correspondiente a dicha entidad de un modo muy sencillo.

Otro de los formatos disponibles es JSON. Este formato tiene como gran ventaja respecto al formato XML que no necesita incluir las etiquetas que definen cada elemento y esto redunda en un tamaño de fichero para la misma información mucho menor. Dada la cantidad de peticiones que realiza la aplicación de escritorio sería conveniente utilizar este formato. Se reduciría la utilización del ancho de banda en gran medida. Sin embargo, se descartó el uso de este formato ya que tanto el soporte en Java como el soporte en Ruby era bastante reducido a la hora de diseñar la aplicación, Octubre de 2007. Por otro lado, a Septiembre de 2008 el soporte en el propio *framework* de Rails ha aumentado por lo que el paso a trabajar con el formato JSON podría llevarse a cabo como trabajo futuro.

Aún estando disponibles las peticiones para las 5 entidades solo utilizaremos 3 de ellas: `Catalog`, `Track` y `Playlist`. Para crear una canción (`Track`) se aportará información relativa al nombre del artista y al título del álbum al que pertenece, por lo que en caso de que fuera necesario crear la entidad `Artist` o `Album` sería la aplicación web la que las crearía a partir de la información proporcionada en la petición para crear la canción.

Para la comunicación y envío de información a la aplicación web se ha utilizado la siguiente biblioteca desarrollada también por Apache: *Jakarta HttpComponents*[21], en concreto utilizando la clase Java

[20] Estas peticiones no se detallaron en la sección 3.3.2 ya que estas serán utilizadas solo por la aplicación de escritorio
[21] *Jakarta HttpComponents* - http://hc.apache.org/httpclient-3.x/

`HttpClient` la cual representa un cliente HTTP, lo que se necesita para comunicarse con la aplicación web. También se estudió la utilización de un *framework* de Java específico para crear aplicaciones REST denominado Restlet[22]. Sin embargo, la utilización de este *framework* y sus bibliotecas se descartó dada su mayor complejidad respecto al componente de comunicación de Apache.

Para el envío de la información seleccionada por el usuario y ya almacenada en el objeto `Library` se seguirán los siguientes pasos:

- En primer lugar es necesario recuperar el identificador del `catalog` en el que se almacenará la información en la aplicación. Este `catalog` será creado en el caso de que el usuario no cuente con uno ya creado en la aplicación web a partir de la información de autenticación proporcionada. Al crear el `catalog` la aplicación web responderá con el identificador asociado a ese `catalog`. En el caso de que ya exista un `catalog` del tipo especificado, simplemente se recuperará su identificador.

- Posteriormente, a partir del objeto `Library`, se obtiene cada una de las listas de reproducción seleccionadas por el usuario para su envío, y para cada una de ellas se envían las `tracks` que la forman. Igualmente, al crearse la canción, la aplicación web devolverá su identificador. Este identificador permitirá posteriormente asociar (añadir) la canción a su lista de reproducción.

  - Una vez se han enviado todas las `Tracks` de una lista de reproducción. Se procede a recuperar el identificador del `Playlist` asociado, creando una nueva o recuperándolo si ya existe en la aplicación web.
  - Posteriormente se añadirán las `tracks` que se crearon correctamente al `Playlist`, al que pertenecen a partir de los identificadores anteriormente almacenados.

- Una vez se ha completado el envío de toda la información se creará un fichero de registro para permitir la sincronización. El diseño de la sincronización será detallado a continuación.

Para resumir, una vez se completa el envío de la información seleccionada por el usuario a la aplicación web se almacena un fichero que contendrá de forma estructurada todo el contenido enviado a la aplicación. Este fichero será utilizado en las tareas de sincronización.

Sincronización

Como se comentó en el apartado 3.4.1, el usuario podrá seleccionar dos opciones de sincronismo: «Borrar y cargar todo» y «Sincronización con el servidor». En este apartado se indican los pasos diseñados para permitir esta funcionalidad. También se comentó que en el caso de no seleccionar ninguna opción de sincronismo se realiza una sincronización con los ficheros de registro, si es que existen en el mismo directorio donde se ejecuta la aplicación. Por lo tanto, existen tres escenarios de sincronismo cuyas fases se detallan a continuación.

- **Sin selección de opción.** En primer lugar se comprueba si existen ficheros de registro de anteriores envíos de información. Si existen se transformará el contenido de dicho fichero en un objeto `Library` que denominaremos `old_library`. Posteriormente se comprobará que información es nueva o ha sido modificada respecto al objeto `old_library`, y será esta la única información que será enviada. Por último, se integrará la información de registro de la última carga de datos con el fichero de registro anterior en el caso de que existiera.

[22] Para ampliar información http://www.restlet.org/

- **Sincronización con el servidor.** Esta opción se recomienda cuando no se conoce en que estado esta la colección musical en local respecto a la colección musical en la aplicación web. También será útil en el caso de que se halla eliminado el fichero de registro. En primer lugar se realizará una petición a la aplicación para descargar toda la información perteneciente al `catalog` del usuario. Esta información será transformada en un fichero de registro que será almacenado en el directorio de ejecución. A partir de aquí la situación es la misma que en el caso de no haber elegido opción de sincronismo.
- **Borrar y cargar todo.** Esta opción será útil para restablecer un estado válido. En primer lugar se eliminará de la aplicación web toda la información del `catalog` específico. También se eliminarán ficheros de registro existentes. Seguidamente se creará el nuevo `catalog` al que se añadirá toda la información. A partir de aquí la situación es la misma que en el caso de no haber elegido opción de sincronismo.

En el apartado 4.3.3 se detallarán los aspectos fundamentales de la implementación del sistema de comunicación de la aplicación de escritorio con la aplicación web. También se detallarán los aspectos relacionados con el sincronismo explicados anteriormente como son: la estructura de los ficheros de registro, la estructura de la información del `catalog` descargado de la aplicación web, la estructura del objeto `Library`, etcétera. Y como se realiza la transformación entre ficheros de texto XML y objetos Java.

CAPÍTULO 4

IMPLEMENTACIÓN

En este capítulo se explicará, aunque no de manera pormenorizada, la implementación de cada una de las partes en que se divide esta aplicación. Se explicarán los aspectos de implementación más relevantes. Ciertos aspectos de la aplicación no se comentarán, ya que se cree que son aspectos intrínsecos al conocimiento de las diferentes tecnologías y también debido a la limitación en la extensión de este documento.

En primer lugar, se explicará como implementar, a grandes rasgos, el diseño sugerido para almacenar la información musical que proporcionan los usuarios, enumerando los aspectos clave, enfocado sobre todo a la generación del modelo de entidades y al establecimiento de la estructura REST de la aplicación. La estructura REST implica fundamentalmente la generación de la estructura de direcciones que manejará la aplicación.

Posteriormente, se explicará cómo se ha implementado el sistema de gestión de usuarios de la parte web de la aplicación. Como ya se ha comentado a la hora del diseño, el sistema se construye a partir de un *plugin*, por lo que solo se comentarán aspectos de puesta en marcha y de configuración.

Más adelante, se expone la implementación de la aplicación de escritorio. Se comentarán cada una de las partes que la forman: interfaz de usuario, recuperación y procesado de la información musical de los usuarios, comunicación con la aplicación web y la sincronización entre ambas aplicaciones. Hasta este punto se encuentra implementado el sistema de almacenamiento de información musical de los usuarios.

Seguidamente, se comentará cómo se ha realizado la consulta a la diferentes APIs a las que tiene acceso la aplicación, exponiendo fundamentalmente los aspectos relativos a la consulta de los diferentes servicios. En general, no se comentarán aspectos relativos a cómo la información recuperada es mostrada en la aplicación web[1].

Por último, y después de haber detallado los aspectos relativos al diseño del sistema de caché en el capítulo anterior, se expone a través de un ejemplo concreto la implementación de este sistema. Se muestra cómo se integran las peticiones bajo demanda y los procesos de fondo. Sólo se expone la implementación de un servicio concreto, ya que todo el conjunto de servicios comparten una implementación muy similar.

[1] Esta información puede ser consultada en el soporte electrónico proporcionado sobre este trabajo

4.1. Información musical de usuarios

En esta sección se abordará a grandes rasgos la implementación del diagrama de entidades. En concreto, la parte del diagrama relacionada con la información musical que proporcionarán los usuarios. Enlazando con el diagrama de entidades se plantea la implementación de la estructura del espacio de direcciones que se utilizarán para proporcionar el carácter REST a la aplicación, y con ello poder manipular los recursos de información.

En primer lugar, se explica la implementación del diagrama de entidades que permite el almacenamiento de la información proporcionada por el usuario y que fue presentado en el aparatado 3.2.

En las aplicaciones Ruby on Rails los atributos de cada una de las entidades se definen en ficheros de migración, y la relación entre entidades se definen en los ficheros que representan cada uno de los modelos (entidades). Los ficheros de migraciones se encuentran en el directorio db/migrate y son los ficheros que crearán la estructura de la aplicación a nivel de base de datos. Para más información sobre los ficheros de migración puede consultarse la documentación sobre ActiveRecord::Migration[2].

Los ficheros que representan a cada una de las entidades o modelos se alojan en el directorio app/models. En estos ficheros se especifican las relaciones entre modelos y las validaciones de cada uno de ellos. Para más información sobre los ficheros de especificación de modelos, se puede consultar la documentación sobre ActiveRecord::Base[3].

Ficheros de migraciones

En la página 120 del apéndice C se muestran los ficheros de migraciones de las entidades que conforman la estructura que almacenará la información musical del usuario.

Como se puede ver, los ficheros de migraciones son autoexplicativos. Simplemente se definen las tablas de datos con las columnas (atributos) que se quieren utilizar y los valores por defecto.

Ficheros de entidades

En la página 121 del apéndice C se muestran las partes de los ficheros que definen las relaciones entre entidades y sus validaciones para las entidades que almacenan la información proporcionada por el usuario. El código fuente completo de estos ficheros se encuentra en el soporte electrónico que adjunta este trabajo.

Igualmente, como ocurría con las migraciones, estos ficheros se explican por si solos. Tanto el apartado de validaciones como el de relaciones entre entidades.

Estructura de direcciones

Para establecer la estructura de direcciones necesaria para el acceso y manipulación de los recursos de información de la aplicación simplemente es necesario definir la estructura en el fichero config/routes.rb. Para generar las rutas comentadas en el apartado de diseño correspondiente (3.3.2) se necesita lo siguiente:

config/routes.rb

```
ActionController::Routing::Routes.draw do |map|
  ...
  map.resources :catalogs do |catalog|
    catalog.resources :artists
    catalog.resources :albums
```

[2] ActiveRecord::Migration - http://api.rubyonrails.org/classes/ActiveRecord/Migration.html
[3] ActiveRecord::Base - http://api.rubyonrails.org/classes/ActiveRecord/Base.html

```
    catalog.resources :tracks
    catalog.resources :playlists,
      :member => { :add_track => :post,
      :remove_track => :delete}
  end
  ...
end
```

Con la anterior definición se generarán los métodos necesarios para la creación de las direcciones de acceso y modificación que se necesitan para satisfacer las necesidades de diseño. Para ampliar información sobre la generación de rutas, puede consultarse la documentación sobre ActionController::Routing[4].

Como ya se comentó Ruby on Rails es un *framework* de desarrollo web que sigue el paradigma Modelo Vista Controlador (MVC). Pero aún no se ha comentado nada acerca de las vistas ni de los controladores. A la hora de generar una nueva entidad en Rails, es posible crear de forma automática los siguientes componentes: fichero de migración, fichero de entidad (sin relaciones ni validaciones), ficheros de vistas y el controlador. A continuación se muestra un ejemplo de cómo se puede crear la entidad catalog:

ruby script/generate scaffold catalog type_catalog:string user_id:integer

Con lo anterior se crean las vistas y también un controlador con las acciones básicas que presenta todo recurso de una aplicación orientada a REST: *index*, *show*, *new*, *create*, *edit*, *update* y *delete*. También se generan el fichero de migración y el fichero de entidad junto con ficheros de pruebas.

Del anterior modo se crearon las entidades tratadas en esta sección. Posteriormente, fueron modificadas tanto las vistas como los controladores para incluir las relaciones que existen entre los diferentes modelos, y que no es posible especificar en la ejecución del anterior comando[5].

4.2. Autenticación, autorización y registro de usuarios

En el apartado de diseño del sistema de gestión de usuarios se propuso la utilización de un *plugin* que construye el sistema de forma prácticamente automática. Esto es debido a que la tarea de creación de un sistema de gestión de usuarios es algo que debe hacerse casi en la totalidad de las aplicaciones, de ahí que existan multitud de *plugins* para proporcionar dicha funcionalidad común. En este apartado de implementación, al tratarse de un sistema que podríamos denominar como estándar, no se detallará la implementación, sino que sólo se comentarán ciertas referencias o comentarios que ayudan a generar la implementación del sistema.

Al tratarse de un diseño de aplicación orientado a REST el *plugin* a utilizar se denomina Restful Authentication[6]. Durante el desarrollo de este trabajo han surgido nuevas versiones de este *plugin*, aunque conservan la misma funcionalidad. Pero aún se conserva la versión utilizada en el desarrollo de la aplicación, y denominada versión *classic*, accesible a través de la misma dirección web.

Como se dijo en el apartado de diseño el sistema de gestión de usuarios fue diseñado e implementado en dos fases. La primera de ellas con una funcionalidad más limitada. Por lo tanto, a la hora de explicar la implementación del sistema se comentará igualmente como se generó la primera iteración, y posteriormente, cómo se completó el sistema con la segunda iteración.

Primera fase. Sistema de gestión de usuarios

En la página web del *plugin* se detallan todos los pasos de instalación y configuración. Por lo tanto, se resumen aquí los aspectos más relevantes.

[4] ActionController::Routing - http://api.rubyonrails.org/classes/ActionController/Routing.html
[5] Para obtener más información sobre el uso de *scaffold* consultar [26]
[6] Restful Authentication - http://github.com/technoweenie/restful-authentication

En primer lugar, se descarga el *plugin* y se aloja dentro del directorio *plugins* de la aplicación con el nombre restful-authentication. Con la siguiente instrucción se generarán los ficheros necesarios para crear el sistema de gestión de usuarios:

ruby script/generate authenticated user sessions

Después de la instalación se aconseja añadir lo siguiente en el fichero config/routes.rb para disponer de URLs más sencillas para acceder a las funciones más habituales de un sistema de gestión de usuarios.

config/routes.rb

```
map.signup  '/signup', :controller => 'users',   :action => 'new'
map.login   '/login',  :controller => 'session', :action => 'new'
map.logout '/logout', :controller => 'session', :action => 'destroy'
```

Las rutas anteriores se utilizarán respectivamente para registrar nuevos usuarios, iniciar sesión en la aplicación y cerrar sesión en la aplicación.

Únicamente con los dos pasos anteriores ya se dispone del sistema de gestión de usuarios con la funcionalidad básica: inicio y fin de sesión de usuarios, autenticación, control de acceso mediante filtros que serán ajustados según las necesidades de la aplicación y el registro de nuevos usuarios. Para ampliar información consultar la página de desarrollo del *plugin* en http://github.com/technoweenie/restful-authentication/tree/classic.

Se destaca nuevamente que se utiliza la versión *classic* del *plugin* y no la nueva versión *master*. Esto es debido a que la versión que se encontraba disponible a la hora del desarrollo de la aplicación era la versión *classic*.

Segunda fase. Sistema de gestión de usuarios

Para completar el sistema de gestión de usuarios se utiliza el anterior *plugin*, pero con la opción de incluir la activación por correo electrónico:

ruby script/generate authenticated user sessions –include-activation

Después de la instalación se completa el fichero config/routes.rb con:

config/routes.rb

```
map.activate '/activate/:activation_code', :controller => 'users', :action => 'activate', :activation_code
    => nil
```

la sentencia anterior habilita la ruta para la activación de cuentas mediante un enlace enviado a la dirección de correo electrónico proporcionada por el usuario al registrarse en la aplicación. También se debe incluir en el fichero config/environment.rb lo siguiente:

config/environment.rb

```
config.active_record.observers = :user_observer
```

Y también recomiendan lo siguiente en el fichero models/user_observer.rb:

config/environment.rb

```
class UserObserver < ActiveRecord::Observer
  def after_create(user)
    user.reload
    UserMailer.deliver_signup_notification(user)
```

```
    end
    def after_save(user)
      user.reload
      UserMailer.deliver_activation(user) if user.recently_activated?
    end
  end
```

Hasta aquí se siguió el manual de instalación del *plugin*, pero a la hora de añadir el soporte para OpenID, y para configurar la cuenta de correo a utilizar por la aplicación para enviar los correos de activación de cuentas, actualización de contraseñas , etcétera, se utilizó la documentación encontrada en diferentes páginas de Internet. La principal referencia para esta integración se puede consultar en [27].

Sin embargo, en el caso de que se pretenda diseñar una aplicación base muy completa en cuanto a funcionalidad básica, a fecha de Septiembre de 2008 se recomienda la utilización de aplicaciones base ya desarrolladas y de acceso público. Entre las diferentes aplicaciones base disponibles en GitHub se recomiendan las siguientes:

- OpenID Rails kit[7]. Este kit permite el inicio de una aplicación web que requiera inicios de sesión tanto vía OpenID como inicios de sesión estándar con nombre de usuario y contraseña. Este kit ha sido creado mediante la integración de diferentes *plugins*.
- Restful Authentication Tutorial[8]. Esta aplicación base se corresponde con la evolución de la documentación que se siguió para la integración de OpenID y del sistema de roles y permisos de la aplicación desarrollada. Esta aplicación presenta mucha mayor funcionalidad, como integración de un sistema de invitaciones para los primeros pasos de la aplicación. También presenta un manejo correcto de errores HTTP 404, *plugins* para la depuración de la aplicación, etcétera.

Para la activación de cuentas de usuario se utilizará como servidor de correo electrónico GMail, ya que ofrece un servicio de garantías para el envío de correos electrónicos, reduce la carga de nuestro servidor, y no es necesario un consumo de espacio extra. El *framework* de Rails incorpora el módulo ActionMailer[9] para el envío de correos electrónicos. Sin embargo, el servidor de Google necesita un tipo de conexiones que proporcionen mayor seguridad, autenticación por túneles denominada TLS, por lo que no es posible utilizar este módulo directamente. Pero existe un *plugin* que se denomina action_mailer_tls[10] y que permite la utilización de ActionMailer como si se quisiera utilizar cualquier servidor de correo sin las restricciones de GMail. Los pasos para la puesta en marcha son los siguientes:

Instalar el *plugin* o descargar los archivos que forman el mismo:

ruby script/plugin install http://code.openrain.com/rails/action_mailer_tls/

Una vez instalado, en la carpeta `/vendor/plugins/action_mailer_tls/sample` se encuentran dos ficheros. Copiar smtp_gmail.rb dentro de la carpeta `/config/initializers`, y copiar mailer.yml.sample a la carpeta `/config`, renombrándolo a mailer.yml. Finalmente, se edita este fichero para usar el nombre de usuario y contraseña de la cuenta de correo de Gmail que se vaya a utilizar, se reinicia el servidor, y ya será posible enviar correos a través de Gmail.

Con lo anterior se consigue que ActionMailer presente la siguiente configuración y que utilice smtp_tls.

```
ActionMailer::Base.server_settings = {
  :address => "smtp.gmail.com",
  :port => 587,
  :domain => "micompañia.com",
```

[7] OpenID Rails kit: http://github.com/stympy/openid-rails-kit
[8] Restful Authentication Tutorial: http://github.com/activefx/restful_authentication_tutorial
[9] Más información en http://api.rubyonrails.org/classes/ActionMailer/Base.html
[10] action_mailer_tls: http://github.com/caritos/action_mailer_tls/

```
  :authentication => :plain,
  :user_name => "nombre_de_usuario",
  :password => "contraseña"
}
```

Por supuesto, antes se debió crear una cuenta de correo electrónico en GMail para utilizarla como servidor de correo electrónico. Para la utilización de GMail como servidor de correo electrónico en una aplicación Rails, se consultaron las referencias [28], [29] y la propia página del *plugin*.

El siguiente paso fue incorporar en el sistema de gestión de usuarios la posibilidad de registrarse en la aplicación mediante OpenID, siguiendo el artículo [27]. Aquí sólo se resumen los pasos más significativos.

Primero debemos instalar la gema de OpenID para Ruby.

gem install ruby-openid

Posteriormente se debe descargar o instalar el plugin de autenticación mediante OpenID de la dirección:

http://github.com/rails/open_id_authentication/tree/master

En esta misma dirección se indican los pasos siguientes al igual que en el artículo. El aspecto fundamental es la relación entre los usuarios y su autenticación mediante OpenID, ya que se debe almacenar la URL que identifica el OpenID del usuario, por lo que es necesario añadir una nueva columna a la tabla de usuarios. Esto se puede llevar a acabo añadiendo en el fichero de migración del *plugin* la siguiente sentencia:

add_column :users, :identity_url, :string

Luego se debe integrar el método de autenticación mediante contraseña y nombre de usuarios o mediante la URL de identificación y OpenID. En los artículos que se utilizaron como guía, se añade como solución una condición que indica qué tipo de autenticación utiliza el usuario.

En los tutoriales seguidos después de los pasos fundamentales se propone el código a utilizar en las vistas de la aplicación. Dichas propuestas se utilizaron en un principio pero luego se fueron modificando.

4.3. Aplicación de Escritorio

En los apartados siguientes se detallará la implementación de la aplicación de escritorio desarrollada conforme al diseño descrito en el apartado 3.4. En primer lugar se explicará la implementación de la interfaz de usuario. Posteriormente el procesado de la información musical del usuario. Por último, se comentan los puntos clave de la comunicación entre la aplicación de escritorio y la aplicación web junto con las tareas de sincronización.

4.3.1. Interfaz de Usuario

En cuanto a la interfaz de usuario no se van ha mostrar detalles de su implementación, ya que es una interfaz muy sencilla y sin elementos diferenciadores. Las clases que permiten su ejecución se encuentran en el paquete **musicstoredesktopapp**[11]. Esta interfaz se desarrolló con el entorno de desarrollo NetBeans[12], que proporciona una interfaz gráfica para el desarrollo de las aplicaciones de escritorio, lo que facilitó enormemente el trabajo.

[11] Accesible en el soporte electrónico proporcionado con este documento
[12] NetBeans - http://www.netbeans.org

4.3.2. Procesado información musical del usuario

En este apartado se comentarán los aspectos claves relacionados en como se extrae la información musical del usuario. Como ya se vio existen tres modos diferentes: procesamiento del fichero de biblioteca de iTunes, procesamiento de los ficheros de audio y procesamiento de la base de datos del reproductor Amarok. A continuación se detallan brevemente cada uno de los tres procesamientos.

Procesamiento iTunes

La clase Java en este trabajo que encapsula el procesamiento de ficheros de biblioteca de iTunes (ficheros *PropertyList*) se encuentra en el paquete **org.pacoguzman.catalog.itunes** y se denomina **PropertyListParser.java**. Esta clase se basa en la utilización de la clase **XMLPropertyListConfiguration.java** perteneciente al API *Configuration* de Apache.

La utilización básica para extraer la información del fichero de biblioteca de iTunes se resume en los tres fragmentos de código[13] que se pueden ver en el apéndice C a partir de la página 128. En el primero de ellos se carga y *parsea* el fichero de biblioteca. Se envía a la función `extractPlaylists` el elemento *playlists* para extraer la información de cada una de las listas de reproducción que contiene.

A su vez, dentro de la anterior función se ejecuta la función `extractPlaylistInfo`, que será la encargada de extraer toda la información que compone una lsita de reproducción. Fundamentalmente, extraerá cada una de las canciones que la forman. La extracción de la información de cada canción se realiza con una función especifica.

Procesamiento directo de ficheros de audio

Como ya se vio en el apartado 3.4.2, la extracción de la información musical del usuario se resume en la búsqueda de ficheros de audio y en la ejecución de tres librerías para la extracción de las etiquetas ID3[14] que presenta un fichero de audio. El procesamiento de los ficheros de audio se encapsula en la clase **FilesParser.java** dentro del paquete **org.pacoguzman.catalog.files**.

A partir de la página 129 se muestran fragmentos de código utilizados para la lectura de las etiquetas de los ficheros MP3. El primero de ellos utiliza la biblioteca de código prioritaria (MyID3), y en caso de que no se pueda extraer la información, se utilizarán el resto de bibliotecas (JID3 y jid3lib).

Procesamiento Amarok

La extracción de la información que almacena el reproductor Amarok se realiza accediendo a la base de datos que guarda el reproductor en forma de fichero. Para la conexión y posterior realización de consultas a la base de datos se utiliza un *driver*[15]. Una vez que se accede a la base de datos sólo se necesita consultar la información deseada, y con esa información construir el objeto `library` que representa la información musical del usuario.

La clase **AmarokParser.java** encapsula todo el proceso descrito anteriormente. El código fundamental de dicha clase se puede consultar en la página 130.

[13] No se corresponden al código completo de procesado

[14] ID3 es un contenedor de metadatos usado generalmente junto a ficheros de audio en formato MP3. Permite almacenar información relativa al fichero de audio en el propio fichero como puede ser: título, artista, álbum, número de canción, etcétera.

[15] Consultar el apartado de diseño 3.4.2

4.3.3. Comunicación con aplicación web y sincronización

En el apartado 3.4.3 se comentaron las decisiones de diseño tomadas para la comunicación entre la aplicación de escritorio y la aplicación web. En este apartado dedicado a la implementación de ese diseño se comentarán los aspectos clave relativos a ese diseño y la solución implementada. En algunos casos se mostrarán detalles de implementación.

En primer lugar se explica el procedimiento en el uso de la clase Java `HttpClient` para la comunicación web. Se explicará el uso para la realización de peticiones GET y POST. En el caso de peticiones POST también se explicará como añadir la información que viaja en dicha petición la cual compone los diferentes recursos.

Petición GET

```
HttpClient cliente = new HttpClient();// Se crea un objeto HttpClient
HttpMethod get = new GetMethod();// Se crea el método HTTP correspondiente
URI uri = new URI("url_destino",false);// Se crea la URL destino
get.setURI(uri);// Se asigna la URL destino al método HTTP

// En caso de que el formato de petición sea XML o JSON es necesario establecer
// la cabecera que indica el tipo de contenido
get.setRequestHeader("Content-type", "application/" + formato);

// Método que establece las credenciales para autenticarse frente a la aplicación
cliente.setCredentials();

// Se ejecuta el método sobre el cliente obteniendo el código de estado
int result = cliente.executeMethod(get);
```

Petición POST

```
HttpClient cliente = new HttpClient();// Se crea un objeto HttpClient
HttpMethod post = new PostMethod();// Se crea el método HTTP correspondiente
URI uri = new URI("url_destino",false);// Se crea la URL destino
post.setURI(uri);// Se asigna la URL destino al método HTTP

// Se crea una entidad a partir de un String que contiene la representación
// en formato XML (resourceXML). El segundo parámetro establece el formato
// de la petición en este caso XML. El tercer parámetro indica la codificación
// de carácteres en este caso utf-8. Es la codificación utilizada para generar
// el recurso XML.
RequestEntity entity = new StringRequestEntity(resourceXML,"application/xml", "utf-8");
post.setRequestEntity(entity);

// Método que establece las credenciales para autenticarse frente a la aplicación
cliente.setCredentials();

// Se ejecuta el método sobre el cliente obteniendo el código de estado
int result = cliente.executeMethod(post);
```

En ambos casos se ejecuta sobre el cliente HTTP el método `setCredentials()`. Este método establece las credenciales para autenticarse en la aplicación web mediante el nombre de usuario y contraseña aportados por el usuario de la aplicación de escritorio, ya que es necesario iniciar una sesión en la aplicación para poder llevar acabo este tipo de peticiones.

```
    /**
     * Método que establece las credenciales para la autenticación
     */
    public void setCredentials() {
        Cookie[] cookies = this.getHttpclient().getState().getCookies();
        // Credenciales para la autenticacion solo si no tenemos la cookie
```

```
        if (cookies.length != 1) {
            // Esquema para autenticación básica
            this.getHttpclient().getState().setCredentials(
                    new AuthScope(HOST, new Integer(PORT), REALM),
                    new UsernamePasswordCredentials(username, password));
        }else{
            logger.debug("Utilizando cookies para autenticación");
        }
    }
```

En cuanto al diseño de la sincronización se vio que un aspecto importante es cómo almacenar la información enviada a la aplicación web o cómo recuperarla de la aplicación web e igualmente almacenar una copia en local. Contar con esta información es lo que va a permitir realizar la sincronización sin que sea necesario en todas las ocasiones enviar toda la información desde cero.

La información de envío actual se conservará en un objeto en memoria de la clase `Library`[16]. Por otro lado el registro de la información enviada se almacena en un fichero a partir de la serialización de un objeto `InfoRegisterType`. Y para el caso de que sea necesario recuperar la información de un recurso `catalog` de la aplicación web, dicha información se almacenará en un fichero a partir de la serialización de un objeto `CatalogType`. Posteriormente este objeto será transformado en un objeto `InfoRegisterType`, como ya se explicó.

«Serialización» de objetos

Para almacenar la información del objeto `catalog` existente en la aplicación web o la información de registro se utiliza serialización basada en un esquema XML. En concreto el Java API for XML Binding (JAXB). JAXB convierte un fichero formateado en varias clases Java. El diagrama de la figura 4.1 ilustra su funcionamiento.

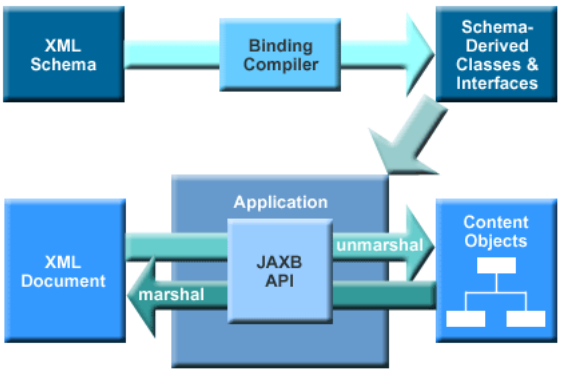

Figura 4.1: Esquema funcionamiento JAXB

Una vez definido el esquema XML, JAXB genera un conjunto de clases Java que leen y escriben instancias de documentos XML correspondientes al esquema XML definido. Con las clases generadas y el API JAXB la aplicación es capaz de:

1. *Marshal* de objetos. Es decir, convertir un objeto Java en un documento XML que puede ser guardado en forma de fichero.

2. *UnMarshall* de documentos XML. La función contraria, es decir, a partir de un documento XML es posible recuperar el objeto Java correspondiente.

[16]Consultar fichero Library.java

Para la utilización de este API se recomienda la consulta de [30, Pág. 275] y del artículo [31].

A continuación se presentan los esquemas XML utilizados en la aplicación de escritorio en un formato gráfico. Para consultar el fichero fuente se deben consultar las siguientes páginas del apéndice C.2.1: 125 y 127.

En primer lugar se muestran los elementos XML y tipos complejos que definen la estructura de la información de registro a serializar (figuras 4.2 y 4.3).

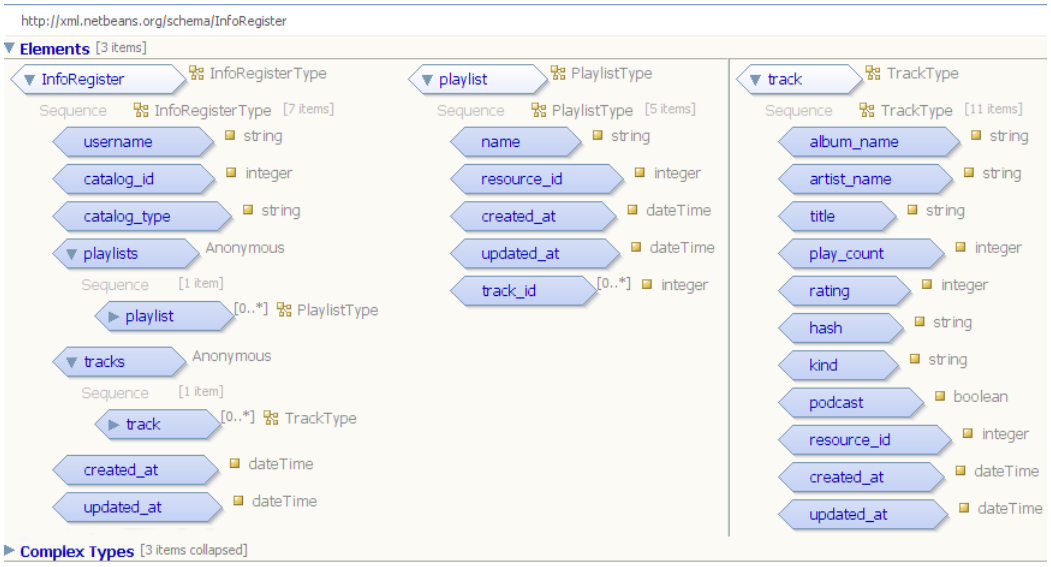

Figura 4.2: Elementos InfoRegisterType

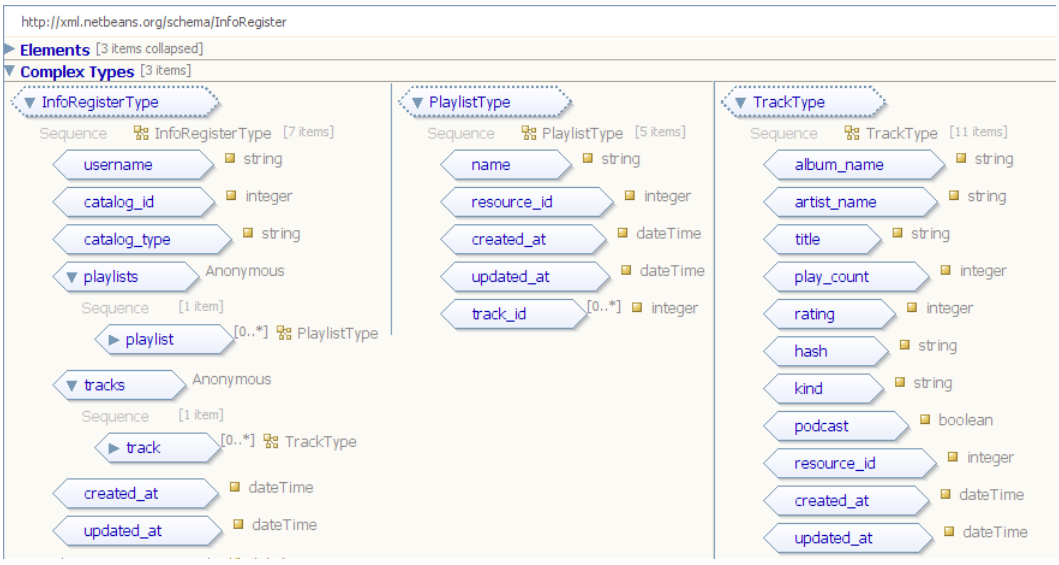

Figura 4.3: Tipos complejos InfoRegisterType

Como se observa sin entrar en detalle en cómo se definen los esquemas XML, el elemento `InfoRegister` contiene información relativa al usuario (`username`), al catálogo (`catalog_id` y `catalog_type`) y está compuesto por el conjunto de canciones (`tracks`) y listas de reproducción (`playlists`). Los elementos `Playlist` contienen la información relativa a una lista de reproducción. Además, el identificador de dicha lista de reproducción en la aplicación web, el elemento `resource_id`. También almacena el

conjunto de identificadores de las canciones que conforman esa lista de reproducción (`track_id`). Por último, el elemento `Track` contiene la información de una canción, además de su identificador en la aplicación web, el elemento `resource_id`.

Para el caso de la información recuperada de la aplicación se muestra la implementación necesaria para que ante una petición XML la aplicación web devuelva toda la información que contiene acerca de un catálogo[17].

```
class CatalogsController < ApplicationController
  ...
  def show
    ...
    respond_to do |format|
      format.html # show.html.erb
      format.xml do
        render :xml => @catalog.to_xml(
          :skip_types => true,
          :include => {
            :tracks => {:skip_types => true,
              :except => [:catalog_id, :artist_id, :album_id, :lyric_id],
              :include => {:artist => {:only => :name},:album => {:only => :name}}
            },
            :playlists => {:skip_types => true,
              :except => :catalog_id,
              :include => {:tracks => {:only => :id}}}
          })
      end
      format.json{ render :json => @catalog.to_json(:include => [:tracks, :playlists])}
    end
  end
  ...
end
```

Como se observa, simplemente es necesario indicar el objeto a devolver en formato XML (@catalog), e indicar los datos que se quiere que se incluyan con `:include`, los que se quiere que no se incluyan con `:except` y con la opción `:skip_types` se indica que no es necesario que se incluyan en las etiquetas XML la información sobre el tipo de dato que contienen. Esto último ayuda a reducir el tamaño que ocupa la información.

A continuación se define el esquema XML utilizado para serializar la información que nos devuelve la aplicación web. Este esquema debe contener toda la información del catálogo solicitado. El esquema fue definido una vez se estableció la estructura que nos devolvía la aplicación web, explicada anteriormente.

Según muestran las figuras 4.4 y 4.5, el esquema XML utilizado para serializar el contenido del catálogo es muy similar al esquema utilizado para serializar la información de registro. Esto es así porque en primer lugar se planteó el esquema de información de registro y posteriormente se pensó en la necesidad de sincronización con el contenido que existiera en el servidor en el caso de que no hubiera almacenada información de registro, o también en aquellos casos en los que se envíe información desde una máquina con una colección musical diferente. Sin embargo, la serialización que realiza la aplicación web no coincidía con el esquema de la información de registro que ya se encontraba implementado y en uso. Por lo tanto, se decidió crear un nuevo esquema e integrar ambos en la aplicación de escritorio, lo que suponía una mayor sencillez y un menor número de cambios.

Las clases Java generadas por JAXB, que se utilizan para la serialización (*marshal* y *unmarshal*) de los objetos, se encuentran disponibles en el soporte electrónico que adjunta este trabajo. En concreto, dentro del área dedicada a la aplicación de escritorio, los paquetes **org.pacoguzman.databinding**.

[17] Código extraído del fichero app/controllers/catalogs_controller.rb

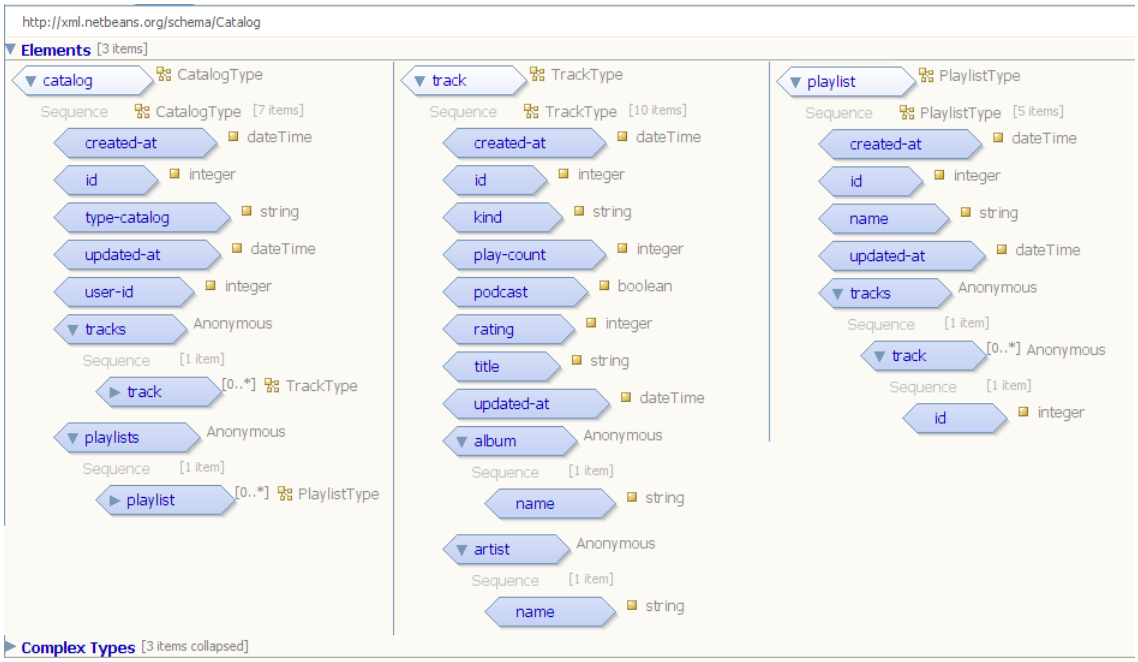

Figura 4.4: Elementos CatalogType

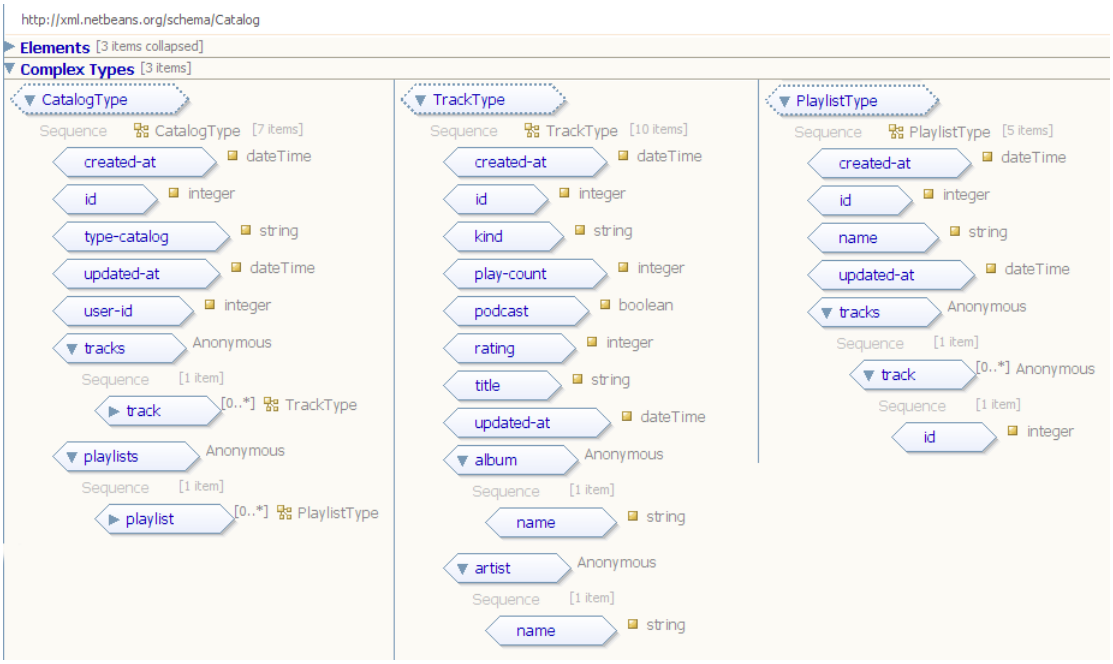

Figura 4.5: Tipos complejos CatalogType

4.4. Implementación Consulta de APIs

En este apartado se detalla cómo se obtiene la información de las distintas fuentes de datos que son consultadas desde la aplicación, pero no se detalla cómo posteriormente poder integrarla en la misma y poder mostrársela al usuario para limitar la extensión de este documento.

En primer lugar se comentarán los servicios procedentes del sistema de Last.fm - Audioscrobbler.

4.4.1. Last.fm - Audioscrobbler

En la siguiente dirección `http://www.audioscrobbler.net/data/webservices/` se pueden consultar todos los métodos proporcionados por este API. El acceso a los servicios se realiza a través de la siguiente URL:

`http://ws.audioscrobbler.com`

Puede realizarse una petición por segundo (promediado a lo largo de una hora). Esto puede llegar a ser una restricción en la aplicación, pero como inicio se cree que es un número suficiente de peticiones disponibles. Todos los servicios son de uso no-comercial y están amparados por la siguiente licencia Creative Commons, Creative Commons Attribution-NonCommercial-ShareAlike License[18].

En los siguientes puntos se enumeran los servicios utilizados en la aplicación de entre todos los que proporciona el API.

Datos de Artista

- *Similar Artists*. Artistas relacionados basado en diferentes fuentes de datos.
- *Top Tracks*. Las canciones más populares de un artista específico.
- *Top Albums*. Los álbumes más populares de un artista específico.
- *Top Tags*. Las etiquetas más populares dadas a un artista.

Datos de Álbum

- *Info*. Información acerca del álbum, incluye el listado de canciones del álbum.

Datos de Canción

- *Similar Tracks*. Canciones relacionadas basado en diferentes fuentes de datos.
- *Top Tags*. Las etiquetas más populares dadas a una canción.

Todos los métodos anteriormente expuestos son accesibles mediante REST y no sería por tanto complicado manejar las respuestas a dichos métodos. Sin embargo, existe una biblioteca de componentes escritos en Ruby denominada Rubyforge.org[19] que pone a nuestra disposición contribuciones *Open Source* de la comunidad de desarrolladores de Ruby. También existe otra gran comunidad de desarrolladores que se ha creado alrededor de un servicio de *hosting*. Esta comunidad se denomina GitHub y es accesible en la dirección http://github.com.

[18] Creative Commons Attribution-NonCommercial-ShareAlike License - http://creativecommons.org/licenses/by-nc-sa/2.0/uk/

[19] RubyForge.org - http://rubyforge.org/

Para la utilización de la API de Last.fm está disponible el siguiente componente desarrollado en Ruby, que se denomina Scrobbler[20] desarrollada por John Nunemaker ([24]). Esta biblioteca será la que se utilice en la aplicación. A continuación se exponen una serie de ejemplos de cómo utilizar la biblioteca y consultar los servicios proporcionados por AudioScrobbler, y cómo mostrar alguno de los datos recuperados.

```
# conseguir artistas similares a uno dado
artist = Scrobbler::Artist.new('Editors', :include_info => true)
artist.similar.each { |a| puts a.name }

# conseguir las canciones más populares de un artista dado
artist = Scrobbler::Artist.new('Editors', :include_info => true)
artist.top_tracks.each { |t| puts t.name }

# conseguir los álbumes más populares de un artista dado
artist = Scrobbler::Artist.new('Editors', :include_info => true)
artist.top_albums.each { |t| puts t.name }

# conseguir la lista de etiquetas asignadas a un artista dado
artist = Scrobbler::Artist.new('Editors', :include_info => true)
artist.top_tags.each { |t| puts t.name }

# conseguir información a cerca de un album dado
album = Scrobbler::Album.new('Editors','The Back Room', :include_info => true)

# conseguir todas las canciones un album
album = Scrobbler::Album.new('Editors','The Back Room', :include_info => true)
album.tracks.each { |t| puts t.name }

# conseguir las etiquetas asignadas a una canción dada
track = Scrobbler::Track.new('Editors','Munich', :include_info => true)
track.tags.each { |t| puts "(#{t.count}) #{t.name} by #{t.artist}" }
```

La aplicación desarrollada se base en estos ejemplos para obtener la información que precisa en cada momento.

4.4.2. LyricWiki

La letra de las canciones se obtienen a través de LyricWiki. Como ya se comentó, este sitio web tiene un servicio web que nos permite recuperar esta información a través de SOAP. Para obtener detalles acerca de las entradas y salidas que requieren y generan cada uno de los métodos/funciones de este servicio, se debe consultar la siguiente dirección web:

http://lyricwiki.org/LyricWiki:SOAP#Allowed_Requests.2FResponses[21]

También está disponible un método alternativo de acceso a través de REST[22], aunque se recomienda el uso a través de SOAP desde el propio dominio.

En principio de esta API se utilizarán exclusivamente los siguiente métodos:

- *checkSongExists(artist, song)*. Este método indica si existe o no la canción del artista especificado en LyricWiki.
- *getSong(artist, song)*. Este método recupera la información asociada a la canción del artista especificado, dicha información incluye la letra de la canción.

El código necesario para acceder a estos métodos se muestra a continuación.

[20] Scrobbler - http://railstips.org/2007/5/11/scrobbler-last-fm-for-ruby o http://github.com/jnunemaker/scrobbler/
[21] Documentación LyricWiki::SOAP - http://lyricwiki.org/LyricWiki:SOAP#Allowed_Requests.2FResponses
[22] LyricWiki::REST - http://lyricwiki.org/LyricWiki:REST

```
require 'scrobbler'

def find_lyricwiki_soap(artist_name, track_title)
  begin
    driver = SOAP::WSDLDriverFactory.new(''http://lyricwiki.org/server.php?wsdl'').create_rpc_driver
  rescue
    return errormsg = 'Failed to establish a connection with LyricWiki.org SOAP Server. LyricWiki.org is
        either down or experiencing an problem with their SOAP server. The script will run, but will be
        less responsive than usual.'
  rescue Timeout::Error
    return errormsg = 'Failed to establish a connection with LyricWiki.org SOAP server. The connection
        timed out. The script will run, but will be less responsive than usual or may not be able to
        contact LyricWiki.org at all.'
  end
  # Chequeamos que exista la canción por si tenemos algún tipo de error
  if driver.checkSongExists(artist_name, track_title)
    @lyrics = driver.getSong(artist_name,track_title).lyrics
  else
    @lyrics = ''This lyric not exists.''
  end

  return @lyrics
end
```

Simplemente se obtiene un *driver* a partir del descriptor del servicio web SOAP(fichero WSDL) y con este *driver* ya es posible acceder a todos los métodos/funciones del API de un modo sencillo como se aprecia en el código mostrado anteriormente.

4.4.3. Amazon - Compras *on-line*

Para recuperar la información acerca de dónde comprar productos musicales se utiliza el API que proporciona Amazon[23], ya que este es unos de los sitios web más importante y con más productos en lo referente a compras a través de Internet.

Para este servicio existe igual que para el acceso a la información de Last.fm un componente desarrollado en Ruby, que se denomina Ruby-AAWS y que ya se presentó en el apartado de diseño correspondiente. Ruby-AAWS facilita la utilización del API de Amazon envolviendo el servicio con una capa de alto nivel desarrollada con Ruby.

En la aplicación se realizarán consultas sobre productos musicales, en concreto a partir del nombre de un artista. Para este tipo de consultas, contando con la biblioteca Ruby-AAWS, es posible obtener la información relativa a los productos disponibles del artista especificado del siguiente modo:

```
def amazon_search(artist)
  # Se establecen los parámetros
  is = ItemSearch.new( 'Music', { 'Artist' => artist.name } )
  rg = ResponseGroup.new( 'Medium' )
  req = Request.new
  req.locale = 'uk'

  # Se realiza la petición
  response = req.search( is, rg, :ALL_PAGES)

  # Mostramos los datos de Amazon más relevantes
  unless response.nil?
    @products = response.products
    puts 'Amazon Search'
    @products.each do |product|
      puts ''#{product[:product_name]} - #{product[:url]}''
    end
  end
end
```

[23] Amazon - http://www.amazon.com

Para poder utilizar este servicio es necesario obtener una clave de acceso a los servicios web, es decir, un identificador (*Key ID*). Es posible obtenerla registrándose en la dirección web Amazon Web Service Account[24]. Como se describe en el fichero introductorio a esta biblioteca, es posible utilizar una cuenta de asociados (*Associates account*), aunque no es estrictamente necesario. Si no se especifica ninguna cuenta de asociado en las invocaciones al servicio a través de la biblioteca se utiliza la cuenta de asociado del creador de Ruby-AAWS. Las cuentas de asociado permiten obtener una comisión a partir de las ventas que se consigan a través de los enlaces que se generen. Los enlaces a los diferentes productos que proporciona este servico de Amazon permiten identificar la cuenta de asociado que los generó y atribuirle una parte de los ingresos por venta que generen.

A partir de la versión 0.4.2 de la gema Ruby-AAWS esta información se centraliza en un fichero de configuración tal y como se indica en el fichero *README* de la biblioteca.

4.4.4. Descargas

En la aplicación también se recupera información acerca de dónde el usuario podría descargar contenido musical mediante programas *peer to peer*. Se ha optado por proporcionar al usuario información sobre archivos *torrents*[25], ya que el protocolo BitTorrent[26] es una de las tecnologías *peer to peer* más utilizadas.

Muchos de los buscadores de archivo *.torrent* más populares permiten la sindicación mediante RSS. La aplicación se va a basar en la lectura de archivos RSS a partir de los cuales se obtiene el contenido que será mostrado al usuario mientras utiliza la aplicación. Se ejemplifica con la consulta al sitio web **Torrentz**, aunque se utilizará este mismo código para acceder a la información del sitio **Mininova.org** con la salvedad de que es necesario modificar las direcciones de consulta.

Torrentz

Para la recuperación de información de enlaces a archivos *.torrent* se utiliza la información que proporciona el sitio web Torrentz[27]. En este sitio web es posible realizar consultas para recuperar información como si se tratara de un lector de noticias (RSS). El formato de consulta sería el siguiente:

http://www.torrentz.com/feed?q=termino+a+consultar

Estas consultas devuelven un fichero RSS como el que se muestra a continuación, que posteriormente será procesado para mostrar al usuario la información relevante.

```
<?xml version''1.0''?>
<rss version=''2.0''>
   <channel>
      <title>Torrentz</title>
      <link>http://www.torrentz.com/</link>
      <description>editors search</description>
      <language>en-us</language>       <item>
         <title>Editors-The Racing Rats-(CDS)-2007-gnvr - music rock music   rock music rock music</title>
         <guid>http://www.torrentz.com/e29e97f5b02481837d132a200410cf1d65b367a5</guid>
         <pubDate>Tue, 04 Dec 2007 08:15:35 +0100</pubDate>
      </item>
      <item>
         <title>editors live in paradiso - video  music videos movies   other other misc movies concerts
              movies   </title>
         <guid>http://www.torrentz.com/77ddc1f7a676becb6c84af73873084250a4bd754</guid>
```

[24] Amazon Web Service Account - https://aws-portal.amazon.com/gp/aws/developer/registration/index.html
[25] Más información en Wikipedia Torrents - http://es.wikipedia.org/wiki/BitTorrent
[26] Especificación Protocolo BitTorrent - http://www.bittorrent.org/beps/bep_0003.html
[27] Torrentz - http://www.torrentz.com

```
        <pubDate>Fri, 09 Nov 2007 14:04:40 +0100</pubDate>
      </item>
      ...
   </channel>
</rss>
```

Y el código implementado para el procesamiento de un RSS mediante Ruby se muestra a continuación:

```
require 'rexml/document'
require 'open-uri'

def run(url, limit = nil)
  doc = open(url).read
  xml = REXML::Document.new(doc)
  @data = {
    :title    => xml.root.elements['channel/title'].text,
    :home_url => xml.root.elements['channel/link'].text,
    :rss_url  => url,
    :items    => []
  }

  xml.elements.each '//item' do |item|
    new_items = {} and item.elements.each do |e|
      new_items[e.name.gsub(/^dc:(\\w)/,"\\1").to_sym] = e.text
    end
    new_items[:pubDate] = DateTime.parse(new_items[:pubDate].to_s)
    @data[:items] << new_items
  end
  return @data
end
```

4.4.5. You Tube - Vídeos Musicales

En la aplicación también se da acceso a vídeos musicales en función de la información que esté consultando el usuario en cada momento. Los vídeos proceden del portal de Internet YouTube.

Para consultar el API de YouTube se utiliza el *plugin* **YouTube Model** desarrollado por Edgar J. Suárez. Este *plugin* está alojado en la siguiente dirección web http://github.com/edgarjs/youtube-model[28].

Este *plugin* posibilita la utilización del nuevo API de YouTube. Como novedad este API permite la subida de vídeos. La utilización se encapsula en un modelo *ActiveResource*[29]. Esta encapsulación permite la utilización del API de un modo muy sencillo.

Un vez se ha instalado el *plugin* en la aplicación Rails y se ha generado un modelo que encapsulará la utilización del API es necesario obtener un clave de cliente y una clave de desarrollador (*Client* y *Developer keys*). Estas claves se pueden obtener en la siguiente dirección web:

http://code.google.com/apis/youtube/dashboard/

En la aplicación se utilizará este *plugin* para el listado de vídeos relacionados con el artista que se consulta. En el caso de denominar el modelo generado como *YouTube* buscar vídeos de un artista se realizaría del siguiente modo:

```
@youtube = YouTube.find(@artist.name)
```

En el caso de querer listar los vídeos de repuesta a la petición anterior se puede utilizar el siguiente fragmento de código:

[28] al menos a la fecha de 22 de Septiembre de 2008
[29] Más información en el API de Rails - http://api.rubyonrails.org/

```
<p><%= image_tag @youtube.logo %></p>
<h1><%= @youtube.title %></h1>
<p><%= "Displaying #{@youtube.startIndex} - #{@youtube.itemsPerPage} of #{@youtube.totalResults}" %></p>
<% for video in @youtube.videos -%>
  <strong><%= video.title %></strong> <small>in <%= video.group.category %></small>
  by <%= link_to video.author.name, "http://youtube.com/#{video.author.name}" %><br />
  <%= simple_format truncate(video.content, :length => 100) %>
  <%= link_to image_tag(video.group.thumbnail.first.url, :alt => video.title), video.link.first.href %><br
      />
  <p> </p>
<% end -%>
```

En la aplicación objeto de este trabajo no se mostrarán los vídeos directamente en la propia aplicación sino que se enlazarán con su ubicación en el portal YouTube. Sin embargo, en el caso de querer mostrar un vídeo contenido en la variable `@video` en la aplicación sería tan sencillo como incrustar el siguiente código en una vista.

```
<h1><%= @video.title %></h1>
<em>by <%= link_to @video.author.name, "http://youtube.com/#{@video.author.name}" %></em>
<p><%= youtube_embed @video %></p>
```

En el caso de querer utilizar alguna de las otras funciones que incluye el *plugin*, como la subida de vídeos al portal YouTube u otras formas de búsqueda, se recomienda consultar la página de desarrollo del *plugin*.

4.5. Implementación del Sistema de Caché

En este último apartado se muestran con un mayor detalle técnico la implementación del sistema de caché diseñado.

Como se ha comentado el sistema de caché presenta un esquema general, que es compartido por la mayoría de los proveedores de contenido a los que la aplicación envía consultas. Por lo tanto, este apartado se centra en la implementación de un servicio proporcionado por LastFM, ya que la implementación para el resto de servicios es muy similar salvo por las diferencias intrínsecas de cada uno de ellos.

Centrándose en el servicio de artistas similares (*similarartists*), en primer lugar se comenta el esquema bajo demanda y posteriormente la utilización de un proceso de fondo. Se comentarán los pasos necesarios y su implementación.

Este capítulo muestra implementaciones utilizando diferentes tecnologías, pero no es necesario que se conozcan con profundidad para entender la idea general que se quiere mostrar, pero si se desea entender mejor la implementación se puede ampliar información de diferentes aspectos en las siguientes referencias:

RJS (Ruby JavaScript). Esquemas visuales para entender RJS en: [32] y [33].

Rails Partials. En los anteriores documentos ya se comentan las plantillas, aunque también es posible consultar la documentación de Rails sobre *partials* en:

- http://api.rubyonrails.org/classes/ActionView/Partials.html

Rake. Es una herramienta de software similar a **make**. Consultar: [34] y [35].

4.5.1. Artistas Similares: bajo demanda

Dentro de la aplicación web el acceso al servicio de artistas similares de LastFM está accesible, junto con otros servicios de LastFM, en las direcciones de la forma:

http://host:port/catalogs/:catalog_id/artists/:id

Como se tiene acceso a diferentes servicios bajo demanda en la misma URL, y no se quiere abandonar ese recurso ni tampoco recargar completamente la página las consultas del usuario, se envían al servidor de la aplicación con una petición *JavaScript* (Paso 1 de 3.3.7). Sólo se actualizará una parte de la página que está siendo visualizada (Paso 8 de 3.3.7).

La petición JavaScript es enviada al siguiente controlador/acción.

http://host:port/catalogs/:catalog_id/scrobbler/:id/similarartists
:controller=>*"scrobbler"*, **:action**=>*"similarartists"*

De esa petición se obtienen dos parámetros:

- **:catalog_id**. Se corresponderá con el identificador de un `catalog`.
- **:id**. Se corresponde con el identificador de un artista de la aplicación.

Dentro de este controlador están implementadas todas las acciones para la consulta de contenido proporcionado por LastFM. Para esta acción concreta se tiene:

scrobbler_controller.rb

```
...
# Artist - Similar artists
def similarartists

  find_similarartists

  respond_to do |format|
    format.js { render :layout => false } #similarartists.js.rjs
  end
end
...
```

La función **find_similarartists** es la encargada de buscar la información solicitada dentro de la aplicación y devolver los artistas similares al controlador si existen en la aplicación, una vez copiada la información en la caché con la función **fill_similar_artists(@artist)**.

scrobbler_controller.rb

```
...
def find_similarartists
  # Se recupera si es necesario la información de LastFM
  InfoArtist.fill_similar_artists(@artist)

  # Se obtiene de la BBDD (caché)
  if @artist.existe_info? and @artist.existen_similares?
    # Se actualiza fecha de consulta de servicio
    @artist.info_artist.update_attributes(:similars_queried_at => Time.now.to_s(:db))
    # Se devuelve el contenido paginado
    @similar_artists = @artist.info_artist.similarartists.paginate(:per_page => 4, :page => params[:page])
  else
    @similar_artists = nil
  end
end
...
```

La función **fill_similar_artists(@artist)** de los modelos **InfoArtist** es la encargada de rellenar (*fill*) la base de datos (caché) con el contenido que nos proporcione LastFM de los artistas similares al artista pasado como parámetro (**@artist**).

info_artist.rb

```
...
def self.fill_similar_artists(artist)
  artist_info = InfoArtist.find_by_name(artist.name)
  if !artist_info.nil?
    if artist_info.info_caducada? || artist_info.similars_caducada?
      # Se obtiene la información de artistas similares desde LastFM (Scrobbler)
      scrobbler_artist = InfoArtist.consultar_scrobbler_para_similares(artist)
      # Se actualiza la información de artistas similares
      artist_info.actualizar_similares(scrobbler_artist) unless scrobbler_artist.nil?
    end
  else
    # Se otbiene la información de artistas similares desde LastFM (Scrobbler)
    scrobbler_artist = InfoArtist.consultar_scrobbler_para_similares(artist)
    unless scrobbler_artist.nil?
      artist_info = InfoArtist.new(:name => scrobbler_artist.name,
        :mbid => scrobbler_artist.mbid,
        :image => scrobbler_artist.image,
        :url_lastfm => scrobbler_artist.url)

      if artist_info.save
        # Se actualiza la información de artistas similares
        artist_info.actualizar_similares(scrobbler_artist)
      else
        logger.info "#{artist_info.errors.full_messages}"
      end
    end
  end
end
...
```

En esta función se comprueba, en primer lugar, si existe alguna información sobre el artista solicitado. Si no existe se consultará LastFM para obtenerla y guardarla en la caché. Además, se actualizarán sus artistas similares (en este caso se crean). Si ya existía contenido en la caché, se comprobará su validez. En caso de que no sea válida, se pedirá y posteriormente se actualizará la caché.

Esta es la función que realmente implementa el sistema de caché para los artistas similares, apoyándose en otras funciones auxiliares (Pasos 3, 4 y 5 de 3.3.7).

Una vez se ha ejecutado la función **find_similarartists** se compone la respuesta mediante RJS y una plantilla (*partial*) y se actualiza la página que está consultando el usuario con el contenido solicitado (Pasos 6, 7 y 8 de 3.3.7).

similarartists.rjs.js

```
page.replace_html :lastfm_tabnav_content,
            :partial => 'scrobbler/artistsimilar',
            :object => @similar_artists
```

_artistsimilar.erb.html

```
<% if @similar_artists %>
  <h2 class="grey_fill">
    <% link_to 'Similar␣Artists', "http://www.lastfm.es/music/#{CGI.escape(@artist.name)}/+similar" %>
  </h2>
  <table>
    <% @similar_artists.in_groups_of(2, false) do |artists| %>
      <tr>
        <% for a in artists %>
          <td>
            <%= link_to(image_tag(a.image, :size => '70x70', :alt => "#{a.name}", :title => "#{a.name}"), a.
                url_lastfm) %>
            <p>
              <%= link_to a.name, a.url_lastfm %><br/>
              <%=h SimilarArtist.find_by_info_artist_id_and_similar_id(@artist.info_artist.id, a.id).match.
                  to_s %><br/>
```

```
            </p>
          </td>
        <% end %>
      </tr>
    <% end %>
  </table>
  <%= will_paginate @similar_artists %>
<% end %>
```

Para resumir, cada servicio de LastFM tiene su correspondiente acción en el controlador **Scrobbler**. Dentro de esa acción se realizará una búsqueda en la aplicación de los datos solicitados (métodos **find_...**). Y dentro de estas funciones se realizaran llamadas sobre métodos de los modelos relacionados con el contenido que se desea consultar, y que son los métodos que implementan la caché. Las primeras funciones permiten que el sistema de caché sea transparente al usuario.

Una vez hemos obtenido el contenido que solicitó el usuario, se compone la respuesta mediante RJS, y en ocasiones con un *partial* (si el contenido a devolver es complejo) actualizando la página de consulta del usuario con el contenido solicitado.

4.5.2. Artistas Similares: proceso de fondo

Para el caso de los procesos de fondo se ha utilizado el software Rake[30]. Este software es muy similar a *make*. Permite ejecutar tareas, descritas en *Rakefiles* (ficheros .rake), las cuales pueden ser programadas en Ruby, que es el lenguaje de desarrollo de esta aplicación. Si además se pretende que estas tareas se ejecuten de forma periódica y automática es posible utilizar tareas programables con Cron (*cronjob*[31]).

Igualmente que para las peticiones bajo demanda, para los procesos de fondo se muestra el servicio de artistas similares. Pero todos los procesos de fondo disponibles tendrían su tarea de fondo que las ejecute y también su tarea específica en un fichero *Rake*, pero la idea es compartida para todos los servicios y sus procesos de fondo.

En primer lugar se debe definir la tarea *rake* a ejecutar. Todas las tareas *rake* de los procesos de fondo residen en el mismo fichero **musicstore.rake** y dentro del espacio de nombres *background*. Para artistas similares:

musicstore.rake

```
namespace :musicstore do
  ...
  namespace :background do
    ...
    desc "Get␣Similar␣Info␣Artists␣from␣LastFM␣for␣artists␣uploaded"
    task :similar_artists_from_artists => :environment do
      # Recuperamos todos los artistas de la base de datos
      artists = Artist.find(:all, :group => 'artists.name')
      artists.each do |a|
        # Para cada uno recuperamos su información de LastFM
        begin

          InfoArtist.fill_similar_artists(a)

        rescue Errno::ECONNABORTED => ex
          puts "Exception␣loading␣similar␣artists␣for␣#{a.name}"
          puts "#{ex.message}"
        rescue EOFError => ex
          puts "Exception␣loading␣similar␣artists␣for␣#{a.name}"
          puts "#{ex.message}"
        rescue Exception => ex
          puts "Exception␣loading␣similar␣artists␣for␣#{a.name}"
```

[30] Consultar - http://www.railsenvy.com/2007/6/11/ruby-on-rails-rake-tutorial
[31] Consulta sobre *cronjobs* - http://www.aota.net/Script_Installation_Tips/cronhelp.php3

```
            puts "#{ex.message}"
          end
        end
      end
      ...
    end
    ...
end
```

Se recorren todos los posibles parámetros de consulta, en este caso, se recorren todos los artistas (modelo `Artist`), y para cada uno de ellos se rellena su información sobre artistas similares (**fill_similar_artists(a)**). Es la misma función que se invoca a través del controlador en el caso de peticiones bajo demanda.

Esta tarea puede ser ejecutada de modo manual en el entorno de desarrollo mediante el siguiente comando:

rake RAILS_ENV=development background:similar_artists_from_artists

Si se pretende que este proceso se lance, por ejemplo, cada lunes a las doce de la noche, se crearía el siguiente *cronjob*:

0 0 * * 1 cd /my_rails_app && rake RAILS_ENV=development musicstore:background:similar_artists_from_artists

CAPÍTULO 5

PRUEBAS Y VALIDACIÓN

En este capítulo se comentarán de forma breve las pruebas y las validaciones realizadas a la aplicación construida, en concreto las realizadas a la aplicación web.

Se utilizan pruebas automáticas que verifican la funcionalidad de la aplicación. Ya que no existe un compilador no es posible conocer errores de sintaxis de antemano, por lo que una buena cobertura de pruebas automáticas es fundamental.

Las aplicaciones Rails, y por lo tanto la aplicación bajo estudio, cuentan con un directorio donde se alojan los ficheros de pruebas que es el directorio *test*. Dentro de este directorio se encuentran otros directorios de los cuales se utilizarán exclusivamente los siguientes: *fixtures*, *unit* y *functional*.

- ***fixtures***. En este directorio se encuentran los ficheros con los datos de ejemplo utilizados en la ejecución de las pruebas.
- ***unit***. En este directorio se encuentran los ficheros ruby que realizan las pruebas de las diferentes entidades/modelos.
- ***functional***. En este directorio se encuentran los ficheros ruby que realizan las pruebas de los diferentes controladores de la aplicación. Las pruebas de los controladores cubren al menos lo siguiente:
  - Comprobar que las variables son asignadas para que sean usadas por la vista correspondiente.
  - Comprobar el código de estado HTTP de las respuestas y su tipo MIME (Multipurpose Internet Mail Extensions).
  - Comprobar que se *renderiza* la vista correspondiente.
  - Comprobar que las redirecciones que se realicen son correctas.
  - Comprobar la asignación correcta de los mensajes *Flash*.
  - Comprobar la modificación en la base de datos como resultado de las acciones.

Para las pruebas se ha utilizado una extensión al *framework* de pruebas que por defecto incluye Rails (Unit::Test) y que se denomina **Shoulda**[1].

Si bien no se ha cubierto todo el código se han realizado un gran número de pruebas como se verá más adelante. La estructuración de las pruebas es muy similar fundamentalmente para las pruebas funcionales de cada uno de los controladores pero también para algunas de las pruebas unitarias de los modelos/entidades. Se expone a continuación una porción del código de pruebas que se realiza al controlador `TracksController` a modo de ejemplo:

[1] Para más información consultar http://www.thoughtbot.com/projects/shoulda o [36]

test/functionals/tracks_controller_test.rb

```
...
logged_in_as(:admin) do
  context "on GET to :index" do
    setup do
      get :index, :catalog_id => catalogs(:catalog_itunes_pacoguzman).id
    end

    should_assign_to :catalog
    should_assign_to :tracks

    should_respond_with :success
    should_render_a_form # search form

    should "has one link to new track page" do
      assert_select "a[href=?]", new_catalog_track_path(catalogs(:catalog_itunes_pacoguzman).id),
        :count => 1, :text => "New track"
    end

    should "has one link to back catalog page" do
      assert_select "a[href=?]", catalog_path(catalogs(:catalog_itunes_pacoguzman).id),
        :count => 1, :text => catalogs(:catalog_itunes_pacoguzman).type_catalog
    end
  end

  context "on GET to :show" do
    setup do
      get :show, :catalog_id => catalogs(:catalog_itunes_pacoguzman).id, :id => tracks(:editors_bullets).
          id
    end

    should_assign_to :catalog
    should_assign_to :track

    should_respond_with :success

    should "has one link to edit page" do
      assert_select "a[href=?]", edit_catalog_track_path(catalogs(:catalog_itunes_pacoguzman),
        tracks(:editors_bullets)),
        :count => 1, :text => "Edit"
    end

    should "has one link back to index tracks page" do
      assert_select "a[href=?]", catalog_tracks_path,
        :count => 1, :text => "Back to index"
    end

    should "has one link back to catalog show page" do
      assert_select "a[href=?]", catalog_path(catalogs(:catalog_itunes_pacoguzman)),
        :count => 1, :text => "Catalog #{catalogs(:catalog_itunes_pacoguzman).id}"
    end
  end
end
...
```

En el anterior extracto de código se establece un primer contexto de pruebas (`logged_in_as(:admin)`). Este contexto indica que el usuario de la pruebas será un usuario con privilegios de administrador. En un nivel inferior se establecen dos nuevos contextos: "`on GET to :show`" y "`on GET to :index`". Estos contextos indican las acciones del controlador `TracksController` que se probarán en cada momento. Dentro de estos contextos se encuentra un método `setup` que se ejecutará para cada una de las pruebas, estableciendo las condiciones de partida de cada prueba. Finalmente las pruebas se corresponden con la palabra reservada **`should`**. Las pruebas se describen directamente de su lectura.

Ahora se expone el código de pruebas unitarias que se realiza sobre la entidad asociada a dicho controlador, la entidad *Tracks*:

test/units/tracks_test.rb

```
...
should_require_attributes :title, :artist, :album, :catalog
should_only_allow_numeric_values_for :rating, :play_count
should_ensure_value_in_range :rating, 0..100

should_belong_to :catalog
should_belong_to :album
should_belong_to :artist
should_have_and_belong_to_many :playlists

should_have_named_scope :recent, :order => 'updated_at DESC'
should_have_named_scope :most_played, :include => [:artist, :album], :order => "play_count DESC"
should_have_named_scope :most_repeated, :select => "COUNT(*) as repeated, tracks.*",
  :group => "lower(tracks.name)", :order => "repeated DESC"
should_have_named_scope :with_lyric, :conditions => "lyric_id IS NOT NULL"

...

context "A New Track" do
  should "be created" do
    assert_difference 'Track.count' do
      track = create_track
      assert !track.new_record?, "#{track.errors.full_messages.to_sentence}"
    end
  end
end
...
```

En este extracto de código de pruebas unitarias, se realizan pruebas sobre las validaciones implementadas al modelo, pruebas sobre las relaciones establecidas entre entidades, también pruebas sobre condiciones de establecimiento de ámbitos (**named_scopes**) y por último se establece un contexto donde se comprueba la creación de una nueva instancia de la clase.

Si se pretende comprobar el funcionamiento de las pruebas es posible ejecutar una serie de comandos que ponen en marcha la ejecución de las pruebas. Para la ejecución de las pruebas de unidad se utiliza el comando **rake test:units** y se obtiene:

217 tests, 412 assertions, 0 failures, 0 errors

Para la ejecución de las pruebas funcionales se utiliza el comando **rake test:functionals** y se obtiene:

280 tests, 377 assertions, 0 failures, 0 errors

En cuanto a la aplicación de escritorio, no ha sido probada mediante la ejecución de pruebas automáticas, pero durante su desarrollo se realizaron pruebas para comprobar su correcto funcionamiento.

CAPÍTULO 6

CONCLUSIONES

PARA resumir el trabajo realizado en este proyecto fin de carrera se exponen a continuación las principales conclusiones alcanzadas en base a los objetivos planteados para este proyecto.

Con la utilización del *framework* de desarrollo Ruby On Rails se ha comprobado que la complejidad de desarrollos de *mashups* y la integración de datos procedentes de fuentes externas es relativamente sencillo. Aún considerando el tiempo de adaptación al *framework* y al lenguaje de programación Ruby parece una mejor solución respecto a la utilización del *framework* Grails. La utilización de Grails suponía mayores desventajas no tanto en lo relativo a su lenguaje de desarrollo Groovy (basado en Java) sino sobretodo en lo relativo a: la juventud del *framework*, menor cantidad de documentación, menor número de desarrollos, menor comunidad de desarrolladores, etcétera.

La complejidad del desarrollo de un *mashup* utilizando Rails es reducida fundamentalmente por las siguientes características:

- Utilización del paradigma MVC. Esta estructura de aplicación web facilita enormemente la creación de un interfaz a la aplicación web.
- Interacción con la base de datos. Como se vio en el apartado 2.2.4.
- Bibliotecas de código disponibles. La existencia de multitud de bibliotecas de código que facilitan el acceso y manipulación de contenido proveniente de diferentes aplicaciones web de contenido musical ya implementadas.
- Comunidad de desarrolladores. Posibilita la consulta de aplicaciones ya desarrolladas y la consulta de dudas que surgen en cualquier etapa de la creación de una aplicación.

En cuanto al caso de estudio se ha construido una aplicación web 2.0 que permite a los usuarios de la misma navegar a través de su biblioteca de información musical desde cualquier navegador con conexión a Internet. Aquí la denominación de aplicación web 2.0 radica en que los contenidos de la aplicación son proporcionados por los usuarios.

La aplicación web se ha estructurado siguendo una orientación *RESTful* lo que permite una organizacion de la información en forma de recursos de información relacionados y direccionables unívocamente. Además, este enfoque *RESTful*, gracias al soporte a dicho enfoque proporcionado por Rails, permite la creación de una interfaz para realizar operaciones sobre los recursos de información que contiene la aplicación. Todo esto permite operar con los recursos de información desde diferentes aplicaciones y no solo desde navegadores web. En este trabajo se utilizada una aplicación de escritorio para enviar y/o sincronizar la información del usuario en la aplicación web.

A su vez este proyecto relaciona la información proporcionada por los usuarios con información procedente de diferentes fuentes de información musical disponibles en la Web. La información obtenida de las diferentes fuentes puede ser por ejemplo: letra de las canciones, imágenes de caratulas de álbumes o imágenes de artistas, productos musicales disponibles en línea, información de descarga, etcétera. La recuperación de diferentes fuentes hace que la aplicación desarrollada se pueda denominar como *mashup*.

Algunos de los proveedores de información consultados presentan limitaciones en el número de consultas que pueden recibir desde una misma aplicación. Lo anterior, junto con la necesidad de proporcionar contenido relevante a los usuarios llevó al diseño y posterior implementación de un sistema de caché para parte de la información consumida.

Contar con la información proporcionada por los diferentes proveedores en un sistema de caché ha permitido la generación de contenido relevante para los usuarios de la aplicación. Esto se consigue relacionando de modo particular la información que proporciona el usuario acerca de su biblioteca musical con la información que se encuentra alojada en el sistema de caché. Este mismo sistema de caché permite mejorar en cierto modo la usuabilidad y la interactivadad de la aplicación web como se ha comentado a lo largo del documento. Otras de las ventajas de su implementación son:

- Reducción de tráfico de red cursado. Si el contenido solicitado ya está en caché y es valido; no es necesario crear nuevas conexiones con los proveedores.
- Reducción número de accesos a proveedores. Si se optimiza la utilización de caché redundará en un menor número de accesos a proveedores.
- Reducción tiempo de respuesta. Todo el contenido que proporciona la aplicación reside en ella misma, contenido de proveedores en caché y contenido proporcionado por usuarios en base de datos.

Uno de los objetivos que no se ha podido alcanzar son las funciones de red social de la aplicación. Esto se ha debido fundamentalmente a la limitación temporal en la realización del proyecto.

CAPÍTULO 7

TRABAJOS FUTUROS

ESTE apartado del documento se enumeran los posibles trabajos futuros a realizar sobre la aplicación presentada con el objetivo de mejorar y/o ampliar sus características.

- Internacionalización. La gran mayoría de las apliciones web 2.0 existentes tiene implementados soportes para la internacionalización. Es decir, es factible la utilización de las aplicaciones en diferentes idiomas. Las versiones futuras del *framework* de Rails, en concreto la versión 2.2, contarán con soporte para internacionalización.
- Sistema de caché.
  - Realizar las peticiones de forma asíncrona a los diferentes servicios anticipándose a posibles peticiones del usuario para mejorar la usabilidad.
  - Realizar la invocación de los diferentes procesos de fondo mientras los usuarios envían la información a la aplicación. Cada vez que se crea nueva información en la aplicación web realizar la invocación de los diferentes procesos de fondo disponibles para completar la información del usuario de forma *sincronizada*. Para implementar este sistema se podría utilizar un sistema de colas donde se fueran encolando los procesos de fondo[1] [2].
- JSON. Utilización del formato JSON para el intercambio de información entre la aplicación web y la aplicación de escritorio. El objetivo es reducir la utilización de ancho de banda durante estos intercambios de información.
- LastFm. Utilización de la nueva versión 2.0 del API de LastFm[3] que incorpora nuevos servicios como por ejemplo: consulta de eventos de artistas.
- Redes Sociales.
  - Implementación de características de red social sobre la aplicación ya implementada. Las características de red social podrían incluir: creación de grupos de usuarios, creación de comentarios entre usuarios valorando a usuarios o a sus bibliotecas de información musical, etcétera.
  - Integración de las características musicales de la aplicación desarrollada en una aplicación base con soporte de redes sociales ya implementada. El desarrollo de una aplicación con sorpote para una comunidad de usuarios o de una red social conlleva un desarrollo complejo por lo

[1] Consultar Startling - http://github.com/starling/starling
[2] Consultar Simplified-Starling - http://github.com/fesplugas/simplified_starling
[3] LastFm API - http://www.last.fm/api/intro

que utilizar una aplicación ya existente es una opción más atractiva que la anteriormente expuesta. De entre las aplicaciones base con soporte para comunidades de usuarios se pueden destacar la siguientes: Community Engine[4], Insoshi[5]y Tog[6]

- Pruebas. Completar el banco de pruebas automáticas de la aplicación web e implementar todo el banco de pruebas automáticas de la aplicación de escritorio.

[4]Community Engine - http://github.com/bborn/communityengine/
[5]Insoshi - http://github.com/insoshi/insoshi
[6]Tog - http://github.com/tog/tog/

APÉNDICE A

MANUAL DE INSTALACIÓN

En este apéndice se indicarán los pasos más importantes para la puesta en marcha de la aplicación bajo estudio en este documento, trasladable a otras aplicaciones Rails, en un entorno de desarrollo.

Para desarrollar cualquier aplicación web con Ruby On Rails deben estar instaladas las siguientes tecnologías:

- **Ruby**. Como Rails es un *framework* escrito en Ruby, es necesario la instalación de Ruby. La aplicación ha sido desarrollada utilizando la versión Ruby 1.8.6, por lo que se recomienda la instalación de dicha versión. En la siguiente dirección web se indica como instalar Ruby en diferentes sistemas operativos:

  http://www.ruby-lang.org/en/downloads/

- **Rails**. Rails es un conjunto de bibliotecas de código que componen el *framework*, en la aplicación de este trabajo utiliza la versión 2.1 de Rails. Para instalar Rails basta con ejecutar el siguiente comando:

  gem install -v=2.1 rails

- **Base de Datos**. Rails es un *framework* diseñado para crear aplicaciones web con almacenamiento de información en base de datos. Por lo tanto, las aplicaciones necesitan para arrancar conexión con una base de datos. En este proyecto se ha utilizado como gestor de base de datos MySQL, en concreto la versión 5.0. Se recomienda la instalación de MySQL como gestor de base de datos para utilizar esta aplicación pero es posible la utilización de otros sistemas gestores de bases de datos simplemente editando el fichero que establece la base de datos a utilizar y que es /config/database.yml (se explica más adelante). Para la descarga de este gestor de base de datos acceder a la siguiente dirección:

  http://dev.mysql.com/downloads/mysql/5.0.html

- **Servidor Web**. Al tratarse del desarrollo de una aplicación web necesitamos un servidor que permita ejecutar la aplicación. Afortunadamente, existen multitud de buenas opciones desarrolladas en ruby para utilizarlas como servidor web. Durante el desarrollo se utilizó **mongrel** como servidor web. También es posible utilizar WEBBrick (disponible al instalar Ruby), Apache o Lighttpd. Para instalar mongrel es necesario instalar la gema correspondiente

  gem install mongrel

Otras tecnologías que se recomienda utilizar pero que no son necesarias para utilizar la aplicación que se ha implementado en este trabajo son:

- **Sistema de Control de Versiones**. En el proyecto se trabajo utilizando SubVersion. Otro sistema de control de versiones que se recomienda es Git. Git es el sistema de control de versiones utilizado para desarrollar Rails.
- **Sistema de despliegue automático de aplicaciones**. En el caso de llevar a producción una aplicación desarrollada en Rails se recomienda la utilización de Capistrano como sistema de despligue automatizado de aplicaciones.

Por otro lado para la puesta en marcha de la aplicación desarrollada en este proyecto es necesario tener instaladas las siguientes bibliotecas y gemas (*gems*) en la máquina en la que sea alojada. En primer lugar se debe añadir un nuevo repositorio de gemas:

gem sources -a http://gems.github.com

- **Hpricot**. Hpricot es un parser HTML flexible escrito en Ruby. En esta aplicación es utilizado por la gema Scrobbler. Versión instalada 0.6.161.

gem install hpricot

- **Scrobbler**. Biblioteca escrita en Ruby para el acceso a algunos de los servicios web disponebles en AudioScrobbler. Versión instalada 0.1.1.

gem install scrobbler

- **Mislav-will_paginate**[1]. Biblioteca utilizada para mostrar un número limitado de registros («paginar»). Versión instalada 2.3.2.

gem install mislav-will_paginate –source=http://gems.github.com

- **ruby-openid**. A Ruby library for verifying and serving OpenID identities. Ruby OpenID makes it easy to add OpenID authentication to your web applications. Versión instalada 2.1.2.

gem install ruby-openid

- **Ruby-AAWS**. Biblioteca escrita en Ruby y empaquetada al final de este trabajo en forma de gema. Es utilizada para la consulta de los servicios web que proporciona Amazon para la versión 4 de su interfaz de servicios web. No confundir con otra gema con nombre Ruby/AWS. Para su instalación consultar:

http://www.caliban.org/ruby/ruby-aws/

- **thoughtbot-shoulda**[2]. Gema que hace fácil la creación, entendimiento y mantenimiento de pruebas.

gem install thoughtbot-shoulda –source=http://gems.github.com

[1] mislav-will_paginate - http://github.com/mislav/will_paginate
[2] thoughtbot-shoulda - http://github.com/thoughtbot/shoulda

- **yfactorial-utility_scopes**[3]. Esta biblioteca aporta una colección de *named_scopes* para su uso con los modelos ActiveRecord.

gem install yfactorial-utility_scopes –source=http://gems.github.com

Una vez se han instalado todas las herramientas necesarias se deben realizar los siguientes pasos para la puesta en marcha de la aplicación:

- Creación de la base de datos para desarrollo y pruebas.

```
mysql –u root –p // Se accede como root o con otro usuario con privilegios
enter password: *******
// Se crean las bases de datos de desarrollo y test
mysql> CREATE DATABASE musicstore_development CHARACTER SET utf8 COLLATE utf8_general_ci;
mysql> CREATE DATABASE musicstore_test CHARACTER SET utf8 COLLATE utf8_general_ci;
```

- Se asignan permisos para el usuario de base de datos que utiliza la aplicación.

```
mysql> GRANT ALL PRIVILEGES ON musicstore_development.* to USUARIO@localhost IDENTIFIED BY '
    CONTRASEÑA';
mysql> GRANT ALL PRIVILEGES ON musicstore_test.* to USUARIO@localhost IDENTIFIED BY '
    CONTRASEÑA';
```

- Se ajusta el fichero config/database.yml con el USUARIO y CONTRASEÑA a utilizar.
- Se ejecuta la siguiente tarea que crea la estructura de base de datos que utiliza la aplicación (dentro del directorio de la aplicación).

rake db:migrate:all

- Se arranca el servidor web en el puerto 8010 (puerto utilizado durante el desarrollo del proyecto).

ruby script/server -p8010

- Por último, Acceder a la aplicación en la siguiente dirección.

http://localhost:8010

[3] yfactorial-utility_scopes - http://github.com/yfactorial/utility_scopes/

APÉNDICE B

MANUAL DE USUARIO

En este apéndice se detallarán algunas de las funcionalidades que proporciona la aplicación implementada mediante imágenes. Las imágenes se corresponden con la utilización de un usuario de la aplicación. Dicho usuario ya ha enviado su información a la aplicación mediante la aplicación de escritorio[1].

Comentarios de las imágenes

Figura B.1. Página de bienvenida una vez el usuario ha iniciado sesión en la aplicación. En esta página se muestran las últimas canciones, listas de reproducción, artistas y álbumes creados o actualizados en toda la aplicación. También es posible obtener información de *LastFm* sobre uno de esos artistas o álbumes. Se muestran productos de *Amazon* de actualidad. Es posible consultar archivos .torrent de contenido musical del día actual de los portales *Mininova* y *Torrentz*.

Figura B.2. Página de detalle sobre un objeto `Catalog`. Se muestran las listas de reproducción, artistas, álbumes y canciones del catálogo que se consulta. Los álbumes, artistas y canciones se muestran *paginados* ya que puede existir un gran número de ellos. Se puede navegar por la colección completa de recursos pulsando los enlaces correspondientes y solo se actualizará la parte correspondiente.

Figure B.3. Página de listado de álbumes del usuario. Se muestra un conjunto de álbumes. También es posible navegar por el conjunto completo con los enlaces al final de la lista. Es posible realizar una búsqueda con sugerencias[2] dinámicas para acceder al detalle de un álbum concreto. En esta página se muestran álbumes recomendados para los artistas que mayor número de veces reproduce el usuario. Esta información se construye gracias a la información proporcionada por LastFm junto con la información que proporciona el usuario. Se muestran hasta cinco álbumes para cada uno de los artistas.

Figure B.4. Página de listado de artistas del usuario. Se muestra un conjunto de artistas. También es posible navegar por el conjunto completo con los enlaces al final de la lista. También es posible realizar una búsqueda con sugerencias para acceder al detalle de un artista concreto (Figura B.7). En esta página se muestran artistas recomendados para los artistas que mayor número de veces reproduce el usuarios. Esta información se construye a partir de información de LastFm e información de la colección del usuario. En la recomendación de un artista en concreto se construyen enlaces donde conseguir más información sobre ese artista. Información en: LyricWiki, Amazon, Google, Torrentz, Yahoo, Yahoo Music y LastFm.

Figura B.5. Página de listado de canciones del usuario. Se muestra un conjunto de canciones y además cómo en los casos anteriores es posible navegar sobre la colección completa o realizar búsquedas para acceder al detalle de una canción (Figura B.8).

[1] No se muestran imágenes de todo el proceso de registro del usuario, descarga de la aplicación de escritorio ni utilización de la misma para no incrementar el tamaño de este documento.

[2] Ver detalle en B.10

Figura B.6. Página de detalle de un álbum. Se muestran datos de la creación del álbum en la aplicación. El conjunto de artistas del álbum en el catálogo del usuario. La lista de canciones del álbum en el catálogo. Y a la derecha se muestra información obtenida de terceros: servicios de LastFm[3] y productos procedentes de Amazon.

Figura B.7. Página de detalle de un artista. Se muestran datos de la creación del artista en la aplicación. El conjunto de álbumes del artista en el catálogo del usuario. La lista de canciones del artista en el catálogo. La lista de otros usuarios (*"fans"*) de dicho artista. Y a la derecha se muestra información obtenida de terceros: servicios de LastFm[4], enlaces a archivos .torrent, vídeos de Youtube.

Figura B.8. Página de detalle de una canción. Se muestran los datos proporcionados por el usuario acerca de la canción junto con enlaces a la página de detalle del álbum y artista de esa canción. Igualmente a la derecha se muestra información obtenida de terceros: servicios de LastFm[5] junto con la letra de la canción obtenida de LyricWiki.

Figura B.9. Página de detalle de una lista de reproducción. Se muestran los datos de la lista de reproducción junto con la lista de las canciones que la componen.

[3]Ver detalle en B.11 - *top albums*, *album info*
[4]Ver detalle en B.12 - *similar artists*, *top albums*, *tags* y *top tracks*
[5]Ver detalle en B.13 - *similar tracks* y *tags*

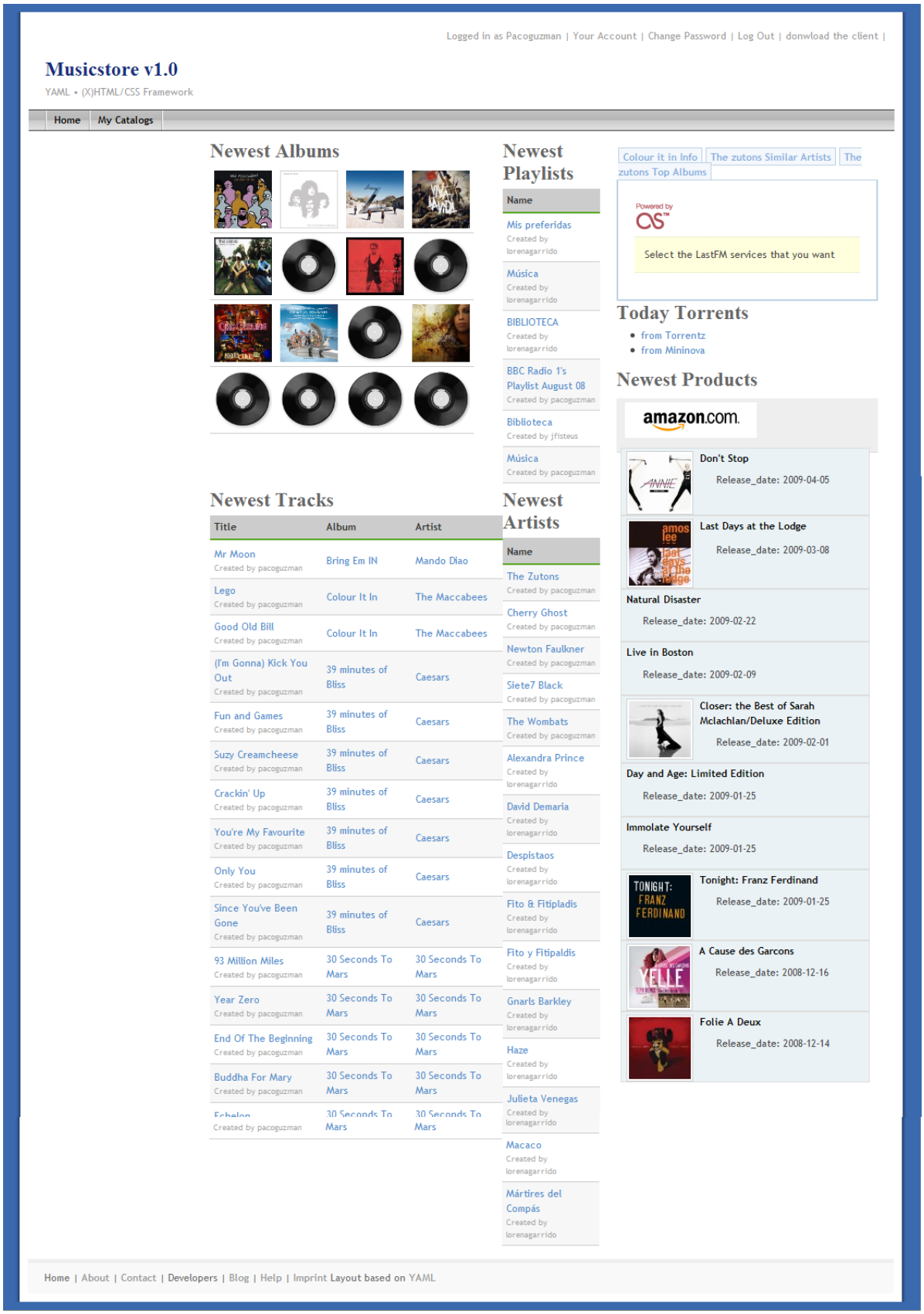

Figura B.1: Página Inicio

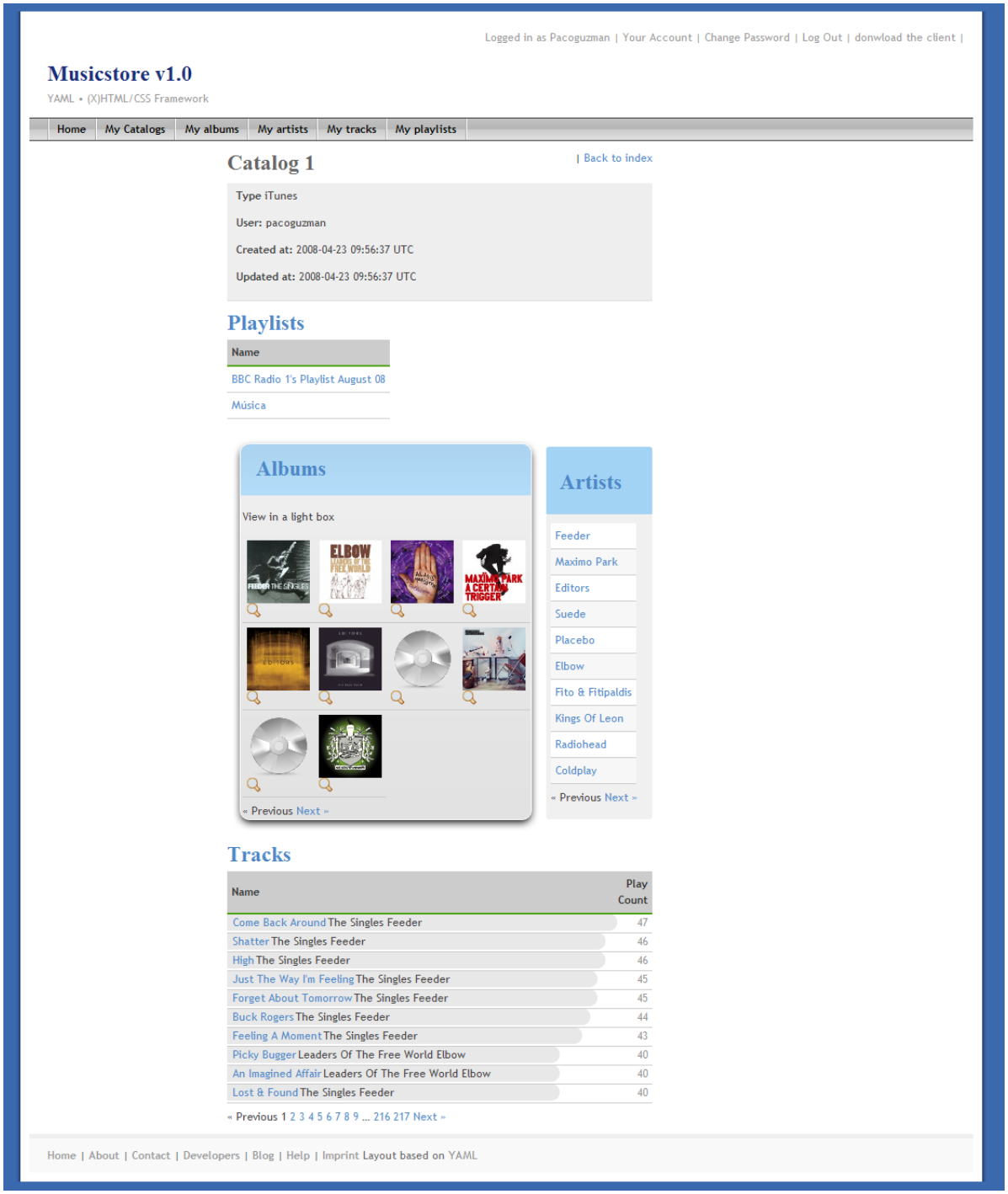

Figura B.2: Detalle `Catalog` (:action =>:show)

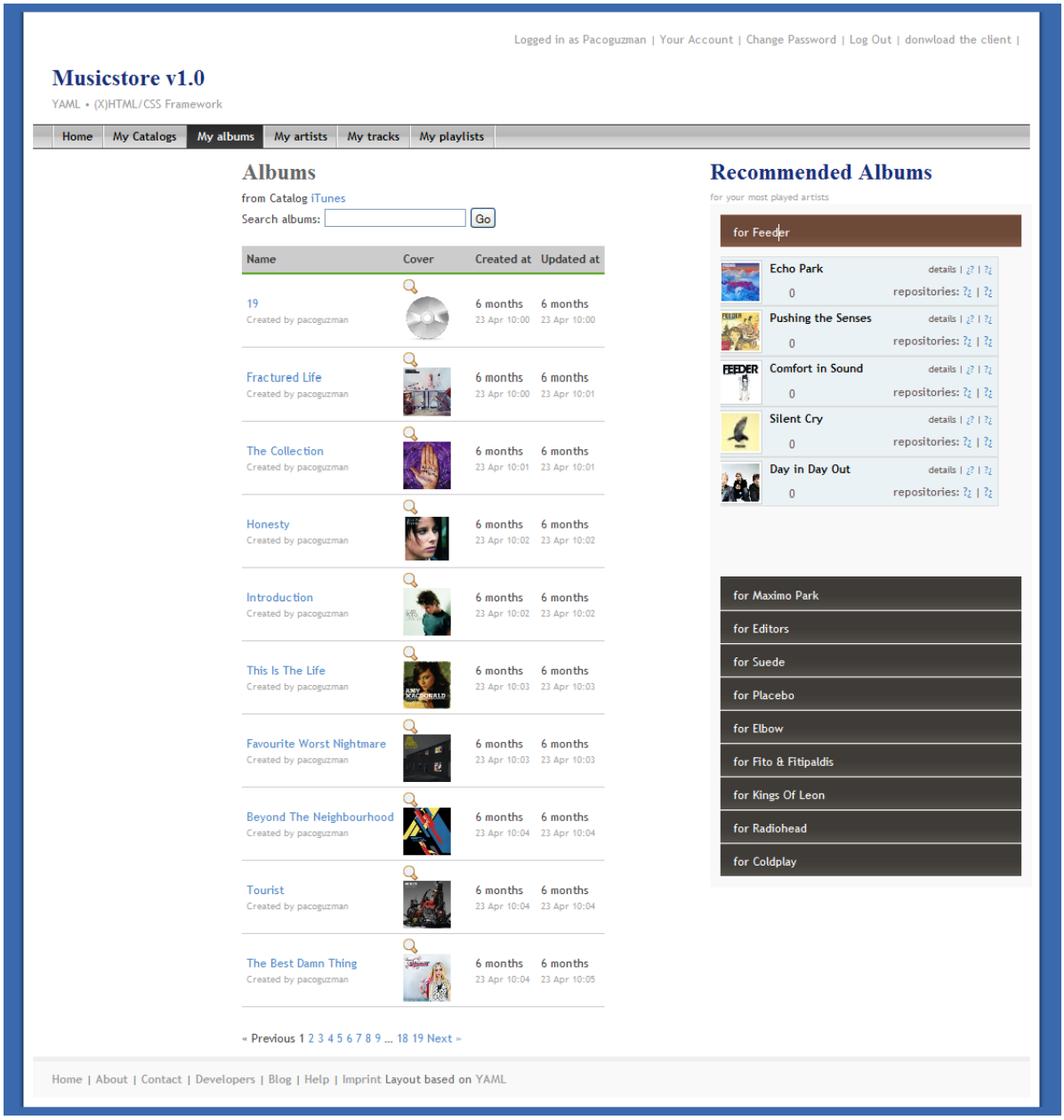

Figura B.3: Listado de Álbumes :action =>:index

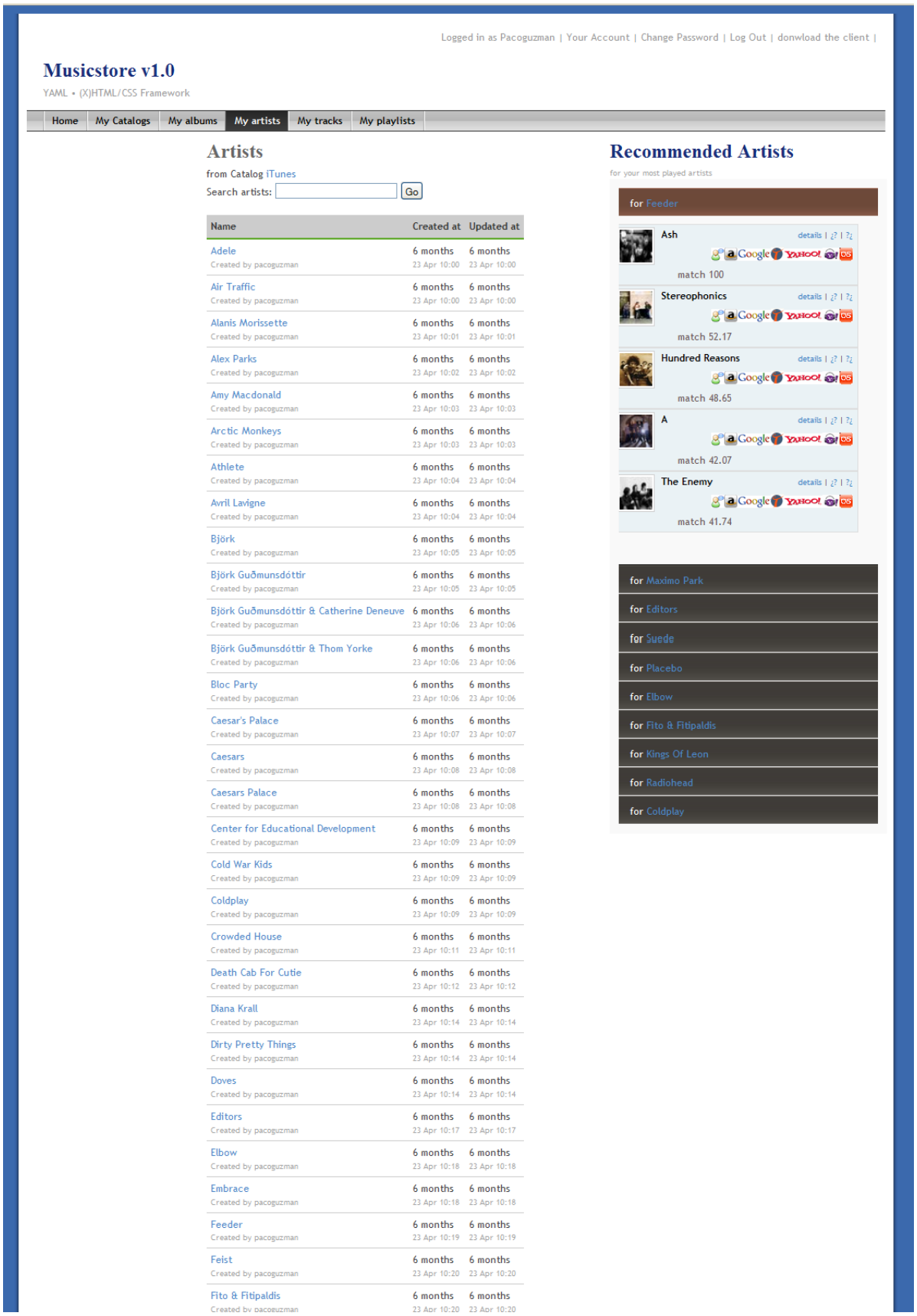

Figura B.4: Listado de Artists :action =>:index

Figura B.5: Listado de Tracks :action =>:index

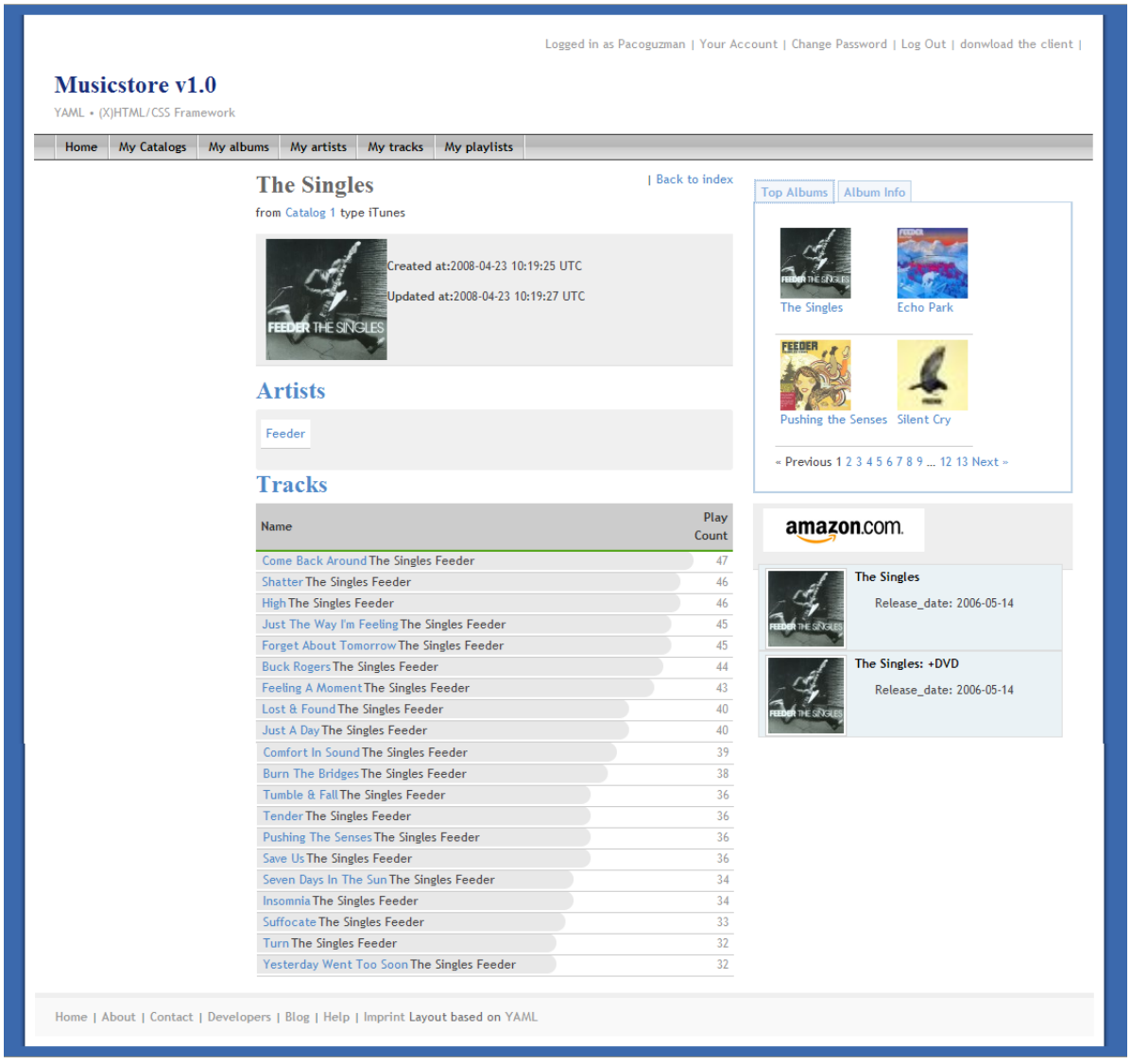

Figura B.6: Detalle de álbum :action =>:show

Figura B.7: Detalle de artista :action =>:show

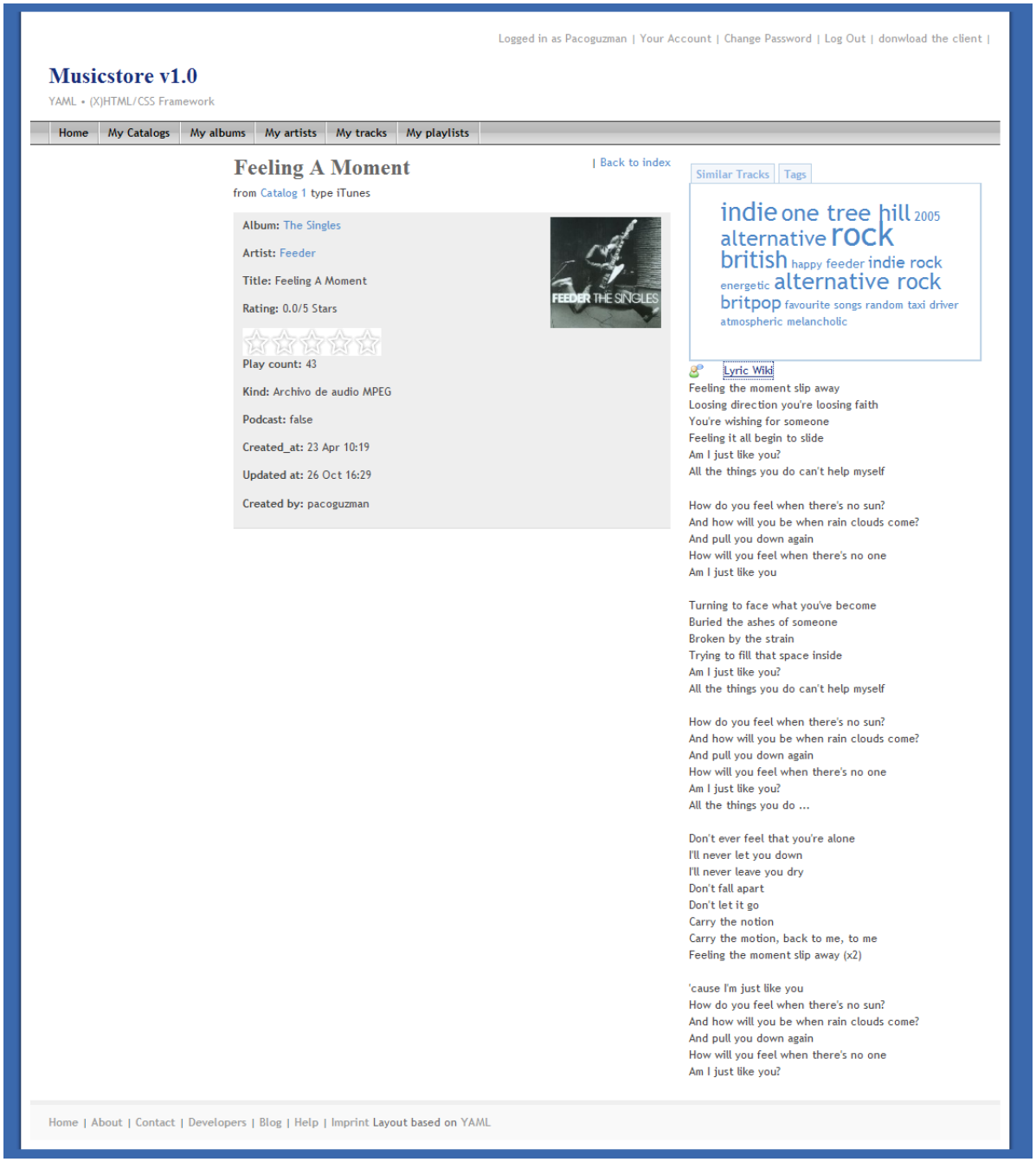

Figura B.8: Detalle de canción :action =>:show

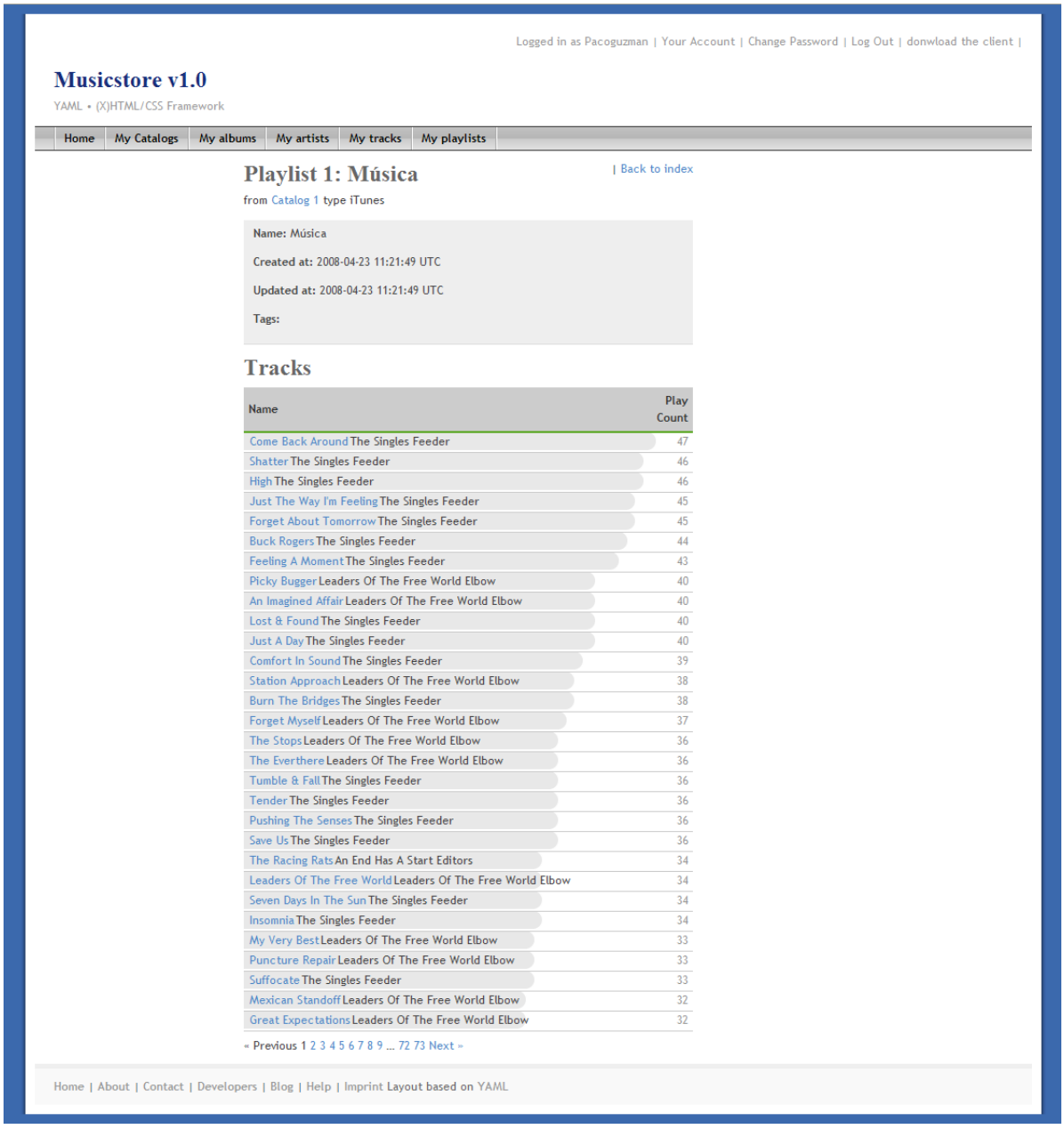

Figura B.9: Detalle de lista de reproducción :action =>:show

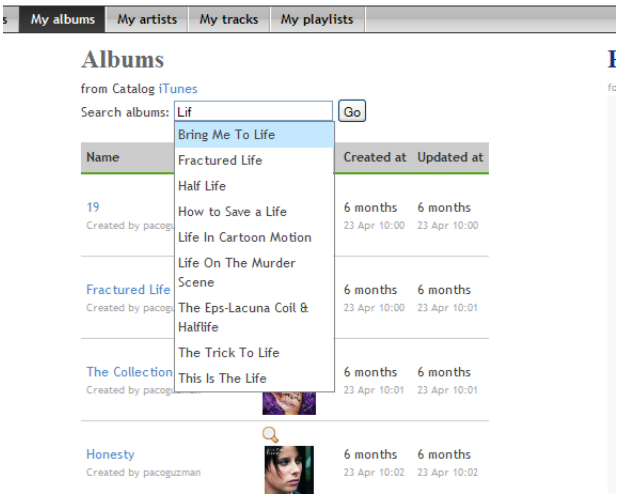

Figura B.10: Detalle de búsqueda con sugerencias

Figura B.11: Detalle de servicios LastFm para un álbum

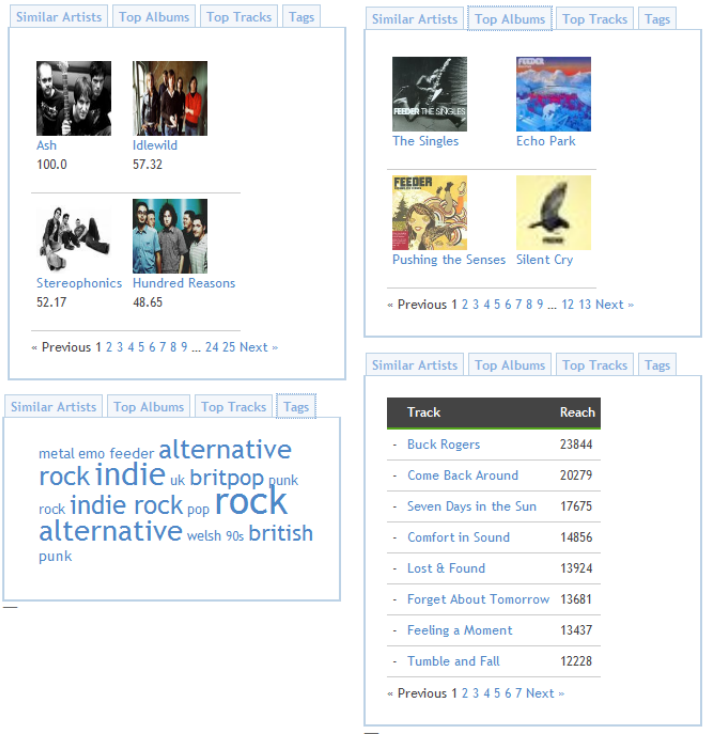

Figura B.12: Detalle de servicios LastFm para un artista

Figura B.13: Detalle de servicios LastFm para una canción

APÉNDICE C

FICHEROS

C.1. Aplicación Web

Ejemplo respuesta Amazon Web Service

```
<?xml version="1.0" encoding="UTF-8"?>
<ItemSearchResponse xmlns="http://webservices.amazon.com/AWSECommerceService/2008-06-26">
    <OperationRequest>
        <HTTPHeaders>
            <Header Name="UserAgent" Value="Ruby/Amazon/AWS 0.4.1">
            </Header>
        </HTTPHeaders>
        <RequestId>1EGSQC878NYCEPJ1Z3P3</RequestId>
        <Arguments>
            <Argument Name="SearchIndex" Value="Music">
            </Argument>
            <Argument Name="AssociateTag" Value="caliban-21">
            </Argument>
            <Argument Name="Artist" Value="Pati Yang">
            </Argument>
            <Argument Name="Service" Value="AWSECommerceService">
            </Argument>
            <Argument Name="ResponseGroup" Value="Medium">
            </Argument>
            <Argument Name="Operation" Value="ItemSearch">
            </Argument>
            <Argument Name="AWSAccessKeyId" Value="1J3AJ28BQ10ATTF0YR02">
            </Argument>
            <Argument Name="Version" Value="2008-06-26">
            </Argument>
        </Arguments>
        <RequestProcessingTime>0.0648150444030762</RequestProcessingTime>
    </OperationRequest>
    <Items>
        <Request>
            <IsValid>True</IsValid>
            <ItemSearchRequest>
                <Artist>Pati Yang</Artist>
                <ResponseGroup>Medium</ResponseGroup>
                <SearchIndex>Music</SearchIndex>
            </ItemSearchRequest>
        </Request>
        <TotalResults>1</TotalResults>
        <TotalPages>1</TotalPages>
        <Item>
            <ASIN>B000ROAJQY</ASIN>
            <DetailPageURL>http://www.amazon.co.uk/Silent-Treatment-Pati-Yang/dp/B000ROAJQY%3FSubscriptionId
                %3D1J3AJ28BQ10ATTF0YR02%26tag%3Dcaliban-21%26linkCode%3Dxm2%26camp%3D2025%26creative%3
                D165953%26creativeASIN%3DB000ROAJQY</DetailPageURL>
            <ItemLinks>
```

```
            <ItemLink>
                <Description>Add To Wishlist</Description>
                <URL>http://www.amazon.co.uk/gp/registry/wishlist/add-item.html%3Fasin.0%3DB000ROAJQY%26
                    SubscriptionId%3D1J3AJ28BQ10ATTF0YR02%26tag%3Dcaliban-21%26linkCode%3Dxm2%26camp%3
                    D2025%26creative%3D12734%26creativeASIN%3DB000ROAJQY</URL>
            </ItemLink>
            <ItemLink>
                <Description>Tell A Friend</Description>
                <URL>http://www.amazon.co.uk/gp/pdp/taf/B000ROAJQY%3FSubscriptionId%3
                    D1J3AJ28BQ10ATTF0YR02%26tag%3Dcaliban-21%26linkCode%3Dxm2%26camp%3D2025%26creative%3
                    D12734%26creativeASIN%3DB000ROAJQY</URL>
            </ItemLink>
            <ItemLink>
                <Description>All Customer Reviews</Description>
                <URL>http://www.amazon.co.uk/review/product/B000ROAJQY%3FSubscriptionId%3
                    D1J3AJ28BQ10ATTF0YR02%26tag%3Dcaliban-21%26linkCode%3Dxm2%26camp%3D2025%26creative%3
                    D12734%26creativeASIN%3DB000ROAJQY</URL>
            </ItemLink>
            <ItemLink>
                <Description>All Offers</Description>
                <URL>http://www.amazon.co.uk/gp/offer-listing/B000ROAJQY%3FSubscriptionId%3
                    D1J3AJ28BQ10ATTF0YR02%26tag%3Dcaliban-21%26linkCode%3Dxm2%26camp%3D2025%26creative%3
                    D12734%26creativeASIN%3DB000ROAJQY</URL>
            </ItemLink>
        </ItemLinks>
        <SalesRank>235612</SalesRank>
        <SmallImage>
            <URL>http://ecx.images-amazon.com/images/I/51uqCMwNU4L._SL75_.jpg</URL>
            <Height Units="pixels">75
            </Height>
            <Width Units="pixels">75
            </Width>
        </SmallImage>
        <MediumImage>
            <URL>http://ecx.images-amazon.com/images/I/51uqCMwNU4L._SL160_.jpg</URL>
            <Height Units="pixels">160
            </Height>
            <Width Units="pixels">160
            </Width>
        </MediumImage>
        <LargeImage>
            <URL>http://ecx.images-amazon.com/images/I/51uqCMwNU4L._SL500_.jpg</URL>
            <Height Units="pixels">500
            </Height>
            <Width Units="pixels">500
            </Width>
        </LargeImage>
        <ImageSets>
            <ImageSet Category="primary">
                <SwatchImage>
                    <URL>http://ecx.images-amazon.com/images/I/51uqCMwNU4L._SL30_.jpg</URL>
                    <Height Units="pixels">30
                    </Height>
                    <Width Units="pixels">30
                    </Width>
                </SwatchImage>
                <SmallImage>
                    <URL>http://ecx.images-amazon.com/images/I/51uqCMwNU4L._SL75_.jpg</URL>
                    <Height Units="pixels">75
                    </Height>
                    <Width Units="pixels">75
                    </Width>
                </SmallImage>
                <ThumbnailImage>
                    <URL>http://ecx.images-amazon.com/images/I/51uqCMwNU4L._SL75_.jpg</URL>
                    <Height Units="pixels">75
                    </Height>
                    <Width Units="pixels">75
                    </Width>
```

```
            </ThumbnailImage>
            <TinyImage>
                <URL>http://ecx.images-amazon.com/images/I/51uqCMwNU4L._SL110_.jpg</URL>
                <Height Units="pixels">110
                </Height>
                <Width Units="pixels">110
                </Width>
            </TinyImage>
            <MediumImage>
                <URL>http://ecx.images-amazon.com/images/I/51uqCMwNU4L._SL160_.jpg</URL>
                <Height Units="pixels">160
                </Height>
                <Width Units="pixels">160
                </Width>
            </MediumImage>
            <LargeImage>
                <URL>http://ecx.images-amazon.com/images/I/51uqCMwNU4L._SL500_.jpg</URL>
                <Height Units="pixels">500
                </Height>
                <Width Units="pixels">500
                </Width>
            </LargeImage>
        </ImageSet>
    </ImageSets>
    <ItemAttributes>
        <Artist>Pati Yang</Artist>
        <Binding>Audio CD</Binding>
        <EAN>0094634179621</EAN>
        <Format>Enhanced
        </Format>
        <Format>Import
        </Format>
        <Label>EMI</Label>
        <ListPrice>
            <Amount>1799</Amount>
            <CurrencyCode>GBP</CurrencyCode>
            <FormattedPrice>Â£17.99</FormattedPrice>
        </ListPrice>
        <Manufacturer>EMI</Manufacturer>
        <NumberOfDiscs>1</NumberOfDiscs>
        <PackageDimensions>
            <Height Units="hundredths-inches">54
            </Height>
            <Length Units="hundredths-inches">555
            </Length>
            <Weight Units="hundredths-pounds">18
            </Weight>
            <Width Units="hundredths-inches">497
            </Width>
        </PackageDimensions>
        <ProductGroup>Music</ProductGroup>
        <ProductTypeName>ABIS_MUSIC</ProductTypeName>
        <Publisher>EMI</Publisher>
        <Studio>EMI</Studio>
        <Title>Silent Treatment</Title>
        <UPC>094634179621</UPC>
    </ItemAttributes>
    <OfferSummary>
        <LowestNewPrice>
            <Amount>992</Amount>
            <CurrencyCode>GBP</CurrencyCode>
            <FormattedPrice>Â£9.92</FormattedPrice>
        </LowestNewPrice>
        <LowestUsedPrice>
            <Amount>798</Amount>
            <CurrencyCode>GBP</CurrencyCode>
            <FormattedPrice>Â£7.98</FormattedPrice>
        </LowestUsedPrice>
        <TotalNew>11</TotalNew>
```

```
                <TotalUsed>2</TotalUsed>
                <TotalCollectible>0</TotalCollectible>
                <TotalRefurbished>0</TotalRefurbished>
            </OfferSummary>
        </Item>
    </Items>
</ItemSearchResponse>
```

Ficheros de migraciones información musical de usuarios

007_create_catalogs.rb

```
class CreateCatalogs < ActiveRecord::Migration
  def self.up
    create_table :catalogs, :force => true  do |t|
      t.integer :user_id
      t.string :type_catalog
      t.timestamps
    end
  end

  def self.down
    drop_table :catalogs
  end
end
```

009_create_artists.rb

```
class CreateArtists < ActiveRecord::Migration
  def self.up
    create_table :artists, :force => true do |t|
      t.integer :catalog_id
      t.string :name, :limit => 255
      t.timestamps
    end
  end

  def self.down
    drop_table :artists
  end
end
```

010_create_albums.rb

```
class CreateAlbums < ActiveRecord::Migration
  def self.up
    create_table :albums, :force => true do |t|
      t.integer :catalog_id
      t.string :name, :limit => 255
      t.string :cover_url, :limit => 255
      t.timestamps
    end
  end

  def self.down
    drop_table :albums
  end
end
```

011_create_tracks.rb

```
class CreateTracks < ActiveRecord::Migration
  def self.up
    create_table :tracks, :force => true  do |t|
      t.integer :catalog_id
```

```
      t.integer :album_id
      t.integer :artist_id
      t.string :title, :limit => 255, :null => false
      t.integer :rating, :maximun => 100, :default => 0
      t.integer :play_count, :default => 0
      t.string :kind, :limit => 255
      t.boolean :podcast, :default => false
      t.timestamps
    end
  end

  def self.down
    drop_table :tracks
  end
end
```

012_create_playlists.rb

```
class CreatePlaylists < ActiveRecord::Migration
  def self.up
    create_table :playlists, :force => true do |t|
      t.integer :catalog_id
      t.string :name, :limit => 255
      t.timestamps
    end
  end

  def self.down
    drop_table :playlists
  end
end
```

014_create_playlists_tracks_table.rb

```
class CreatePlaylistsTracksTable < ActiveRecord::Migration
  def self.up
    create_table :playlists_tracks, :id => false do |t|
      t.integer :playlist_id, :null => false
      t.integer :track_id, :null => false
    end
  end

  def self.down
    drop_table :playlists_tracks
  end
end
```

Ficheros de entidades información musical de usuarios

catalog.rb

```
class Catalog < ActiveRecord::Base
  validates_presence_of :user
  validates_presence_of :type_catalog
  validates_inclusion_of :type_catalog,
    :in => %w( iTunes amarok files ),
    :message => "Should␣be␣'iTunes'␣or␣'amarok'␣or␣'files'"
  # El type_catalog es único a nivel de usuario
  validates_uniqueness_of :type_catalog, :scope => :user_id

  belongs_to :user
  # Se incluye la información del album y del artist
  has_many :tracks, :include => { :album => {}, :artist => {} }, :order => "tracks.play_count␣desc", :
      dependent => :destroy
  has_many :playlists, :order => "id␣desc", :extend => TagCountsExtension, :dependent => :destroy
```

```
  has_many :artists, :dependent => :destroy
  has_many :albums, :dependent => :destroy
  ...
end
```

artist.rb

```
class Artist < ActiveRecord::Base
  validates_presence_of :name
  validates_uniqueness_of :name, :scope => :catalog_id

  belongs_to :catalog
  has_many :tracks, :order => 'tracks.play_count DESC'
  has_many :albums, :through => :tracks, :uniq => true
  ...
end
```

album.rb

```
class Album < ActiveRecord::Base
  validates_presence_of :name
  validates_uniqueness_of :name, :scope => :catalog_id

  belongs_to :catalog
  has_many :tracks, :order => 'tracks.play_count DESC'
  has_many :artists, :through => :tracks, :uniq => true
  ...
end
```

track.rb

```
class Track < ActiveRecord::Base
  validates_presence_of :title
  validates_presence_of :artist, :album, :catalog
  validates_numericality_of :rating
  validates_numericality_of :play_count, :greater_than_or_equal_to => 0
  validates_inclusion_of :rating, :in => 0..100

  belongs_to :catalog
  belongs_to :album
  belongs_to :artist
  belongs_to :lyric
  has_and_belongs_to_many :playlists
end
```

playlist.rb

```
class Playlist < ActiveRecord::Base

  validates_presence_of :name
  # El name es único a nivel de catalog
  validates_uniqueness_of :name, :scope => :catalog_id

  belongs_to :catalog
  has_and_belongs_to_many :tracks
end
```

C.2. Aplicación de Escritorio

Ejemplo fichero de biblioteca iTunes original

iTunes_Music_Library.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.
    dtd">
<plist version="1.0">
<dict>
        <key>Major Version</key><integer>1</integer>
        <key>Minor Version</key><integer>1</integer>
        <key>Application Version</key><string>7.7.1</string>
        <key>Features</key><integer>5</integer>
        <key>Show Content Ratings</key><true/>
        <key>Music Folder</key><string>file://localhost/F:/Documents%20and%20Settings/Javi/Mis%20documentos/
            Mi%20m%C3%BAsica/iTunes/iTunes%20Music/</string>
        <key>Library Persistent ID</key><string>E40D44C9C4AC746F</string>
        <key>Tracks</key>
        <dict>
                <key>356</key>
                <dict>
                        <key>Track ID</key><integer>356</integer>
                        <key>Name</key><string>Capricorn (A Brand New Name)</string>
                        <key>Artist</key><string>30 Seconds To Mars</string>
                        <key>Album Artist</key><string>30 Seconds to Mars</string>
                        <key>Composer</key><string>Jared Leto</string>
                        <key>Album</key><string>30 Seconds To Mars</string>
                        <key>Genre</key><string>Industrial</string>
                        <key>Kind</key><string>Archivo de audio MPEG</string>
                        <key>Size</key><integer>5640381</integer>
                        <key>Total Time</key><integer>233064</integer>
                        <key>Track Number</key><integer>1</integer>
                        <key>Year</key><integer>2002</integer>
                        <key>Date Modified</key><date>2008-04-10T07:41:52Z</date>
                        <key>Date Added</key><date>2007-11-14T06:55:27Z</date>
                        <key>Bit Rate</key><integer>192</integer>
                        <key>Sample Rate</key><integer>44100</integer>
                        <key>Play Count</key><integer>2</integer>
                        <key>Play Date</key><integer>3289902209</integer>
                        <key>Play Date UTC</key><date>2008-04-01T11:43:29Z</date>
                        <key>Persistent ID</key><string>E40D44C9C4AC7485</string>
                        <key>Track Type</key><string>File</string>
                        <key>Location</key><string>file://localhost/F:/Documents%20and%20Settings/Javi/Mis
                            %20documentos/Mi%20m%C3%BAsica/DVD_Grabados/30%20Seconds%20To%20Mars/30%20Second
                            %20To%20Mars%20-%2030%20Seconds%20To%20Mars/01%20-%2030%20Seconds%20To%20Mars
                            %20-%20Capricorn%20(A%20Brand%20New%20Name).mp3</string>
                        <key>File Folder Count</key><integer>-1</integer>
                        <key>Library Folder Count</key><integer>-1</integer>
                </dict>
                <key>357</key>
                <dict>
                        <key>Track ID</key><integer>357</integer>
                        <key>Name</key><string>Edge Of The Earth</string>
                        <key>Artist</key><string>30 Seconds To Mars</string>
                        <key>Album Artist</key><string>30 Seconds to Mars</string>
                        <key>Composer</key><string>Jared Leto</string>
                        <key>Album</key><string>30 Seconds To Mars</string>
                        <key>Genre</key><string>Industrial</string>
                        <key>Kind</key><string>Archivo de audio MPEG</string>
                        <key>Size</key><integer>6708685</integer>
                        <key>Total Time</key><integer>277577</integer>
                        <key>Track Number</key><integer>2</integer>
                        <key>Year</key><integer>2002</integer>
                        <key>Date Modified</key><date>2008-04-10T07:41:52Z</date>
                        <key>Date Added</key><date>2007-11-14T06:55:27Z</date>
                        <key>Bit Rate</key><integer>192</integer>
                        <key>Sample Rate</key><integer>44100</integer>
                        <key>Play Count</key><integer>4</integer>
                        <key>Play Date</key><integer>3291963005</integer>
                        <key>Play Date UTC</key><date>2008-04-25T08:10:05Z</date>
                        <key>Persistent ID</key><string>E40D44C9C4AC7487</string>
                        <key>Track Type</key><string>File</string>
```

```
                        <key>Location</key><string>file://localhost/F:/Documents%20and%20Settings/Javi/Mis
                            %20documentos/Mi%20m%C3%BAsica/DVD_Grabados/30%20Seconds%20To%20Mars/30%20Second
                            %20To%20Mars%20-%2030%20Seconds%20To%20Mars/02%20-%2030%20Seconds%20To%20Mars
                            %20-%20Edge%20Of%20The%20Earth.mp3</string>
                        <key>File Folder Count</key><integer>-1</integer>
                        <key>Library Folder Count</key><integer>-1</integer>
                </dict>
        </dict>
        <key>Playlists</key>
        <array>
                <dict>
                        <key>Name</key><string>BIBLIOTECA</string>
                        <key>Master</key><true/>
                        <key>Playlist ID</key><integer>3040</integer>
                        <key>Playlist Persistent ID</key><string>E40D44C9C4AC7470</string>
                        <key>Visible</key><false/>
                        <key>All Items</key><true/>
                        <key>Playlist Items</key>
                        <array>
                                <dict>
                                        <key>Track ID</key><integer>356</integer>
                                </dict>
                                <dict>
                                        <key>Track ID</key><integer>357</integer>
                                </dict>
                                <dict>
                                        <key>Track ID</key><integer>358</integer>
                                </dict>
                        </array>
                </dict>
        </array>
</dict>
</plist>
```

Ejemplo resultado de procesamiento del fichero anterior

processed.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.
    dtd">
<plist version="1.0">
<dict>
        <key>Major Version</key><integer>1</integer>
        <key>Minor Version</key><integer>1</integer>
        <key>Application Version</key><string>7.7.1</string>
        <key>Features</key><integer>5</integer>
        <key>Show Content Ratings</key><true/>
        <key>Music Folder</key><string>file://localhost/F:/Documents%20and%20Settings/Javi/Mis%20documentos/
            Mi%20m%C3%BAsica/iTunes/iTunes%20Music/</string>
        <key>Library Persistent ID</key><string>E40D44C9C4AC746F</string>
        <key>Tracks</key>
        <dict>
                <key>356</key>
                <dict>
                        <key>Track ID</key><integer>356</integer>
                        <key>Name</key><string>Capricorn (A Brand New Name)</string>
                        <key>Artist</key><string>30 Seconds To Mars</string>
                        <key>Album</key><string>30 Seconds To Mars</string>
                        <key>Genre</key><string>Industrial</string>
                        <key>Kind</key><string>Archivo de audio MPEG</string>
                        <key>Track Number</key><integer>1</integer>
                        <key>Year</key><integer>2002</integer>
                        <key>Play Count</key><integer>2</integer>
                        <key>Persistent ID</key><string>E40D44C9C4AC7485</string>
                        <key>Track Type</key><string>File</string>
                </dict>
```

```
                <key>357</key>
                <dict>
                        <key>Track ID</key><integer>357</integer>
                        <key>Name</key><string>Edge Of The Earth</string>
                        <key>Artist</key><string>30 Seconds To Mars</string>
                        <key>Album</key><string>30 Seconds To Mars</string>
                        <key>Genre</key><string>Industrial</string>
                        <key>Kind</key><string>Archivo de audio MPEG</string>
                        <key>Track Number</key><integer>2</integer>
                        <key>Year</key><integer>2002</integer>
                        <key>Play Count</key><integer>4</integer>
                        <key>Persistent ID</key><string>E40D44C9C4AC7487</string>
                        <key>Track Type</key><string>File</string>
                </dict>
        </dict>
        <key>Playlists</key>
        <array>
                <dict>
                        <key>Name</key><string>BIBLIOTECA</string>
                        <key>Master</key><true/>
                        <key>Playlist ID</key><integer>3040</integer>
                        <key>Playlist Persistent ID</key><string>E40D44C9C4AC7470</string>
                        <key>Visible</key><false/>
                        <key>All Items</key><true/>
                        <key>Playlist Items</key>
                        <array>
                                <dict>
                                        <key>Track ID</key><integer>356</integer>
                                </dict>
                                <dict>
                                        <key>Track ID</key><integer>357</integer>
                                </dict>
                                <dict>
                                        <key>Track ID</key><integer>358</integer>
                                </dict>
                        </array>
                </dict>
        </array>
</dict>
</plist>
```

C.2.1. Esquemas XML para serialización de objetos

Fichero Catalog.xsd

Catalog.xsd

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xml.netbeans.org/schema/Catalog"
            xmlns:tns="http://xml.netbeans.org/schema/Catalog"
            elementFormDefault="qualified">
<xsd:element name="track" type="tns:TrackType"></xsd:element>
    <xsd:complexType name="TrackType">
        <xsd:sequence>
            <xsd:element name="created-at" type="xsd:dateTime"></xsd:element>
            <xsd:element name="id" type="xsd:integer"></xsd:element>
            <xsd:element name="kind" type="xsd:string"></xsd:element>
            <xsd:element name="play-count" type="xsd:integer"></xsd:element>
            <xsd:element name="podcast" type="xsd:boolean"></xsd:element>
            <xsd:element name="rating">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:integer">
                        <xsd:minInclusive value="0"/>
                        <xsd:maxInclusive value="100"/>
```

```
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:element>
        <xsd:element name="title" type="xsd:string"></xsd:element>
        <xsd:element name="updated-at" type="xsd:dateTime"></xsd:element>
        <xsd:element name="album">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="name" type="xsd:string"></xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="artist">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="name" type="xsd:string"></xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="playlist" type="tns:PlaylistType"></xsd:element>
<xsd:complexType name="PlaylistType">
    <xsd:sequence>
        <xsd:element name="created-at" type="xsd:dateTime"></xsd:element>
        <xsd:element name="id" type="xsd:integer"></xsd:element>
        <xsd:element name="name" type="xsd:string"></xsd:element>
        <xsd:element name="updated-at" type="xsd:dateTime"></xsd:element>
        <xsd:element name="tracks" minOccurs="1" maxOccurs="1">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="track" minOccurs="0" maxOccurs="unbounded">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="id" type="xsd:integer"></xsd:element>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="catalog" type="tns:CatalogType"></xsd:element>
<xsd:complexType name="CatalogType">
    <xsd:sequence>
        <xsd:element name="created-at" type="xsd:dateTime"></xsd:element>
        <xsd:element name="id" type="xsd:integer"></xsd:element>
        <xsd:element name="type-catalog">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:pattern value="Files|iTunes|Amarok"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:element>
        <xsd:element name="updated-at" type="xsd:dateTime"></xsd:element>
        <xsd:element name="user-id" type="xsd:integer"></xsd:element>
        <xsd:element name="tracks">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="track" type="tns:TrackType" minOccurs="0" maxOccurs="unbounded"><
                        /xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="playlists">
            <xsd:complexType>
```

```
                    <xsd:sequence>
                        <xsd:element name="playlist" type="tns:PlaylistType" minOccurs="0" maxOccurs="
                            unbounded"></xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Fichero InfoRegister.xsd

InfoRegister.xsd

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xml.netbeans.org/schema/InfoRegister"
            xmlns:tns="http://xml.netbeans.org/schema/InfoRegister"
            elementFormDefault="qualified">
    <xsd:element name="track" type="tns:TrackType"></xsd:element>
    <xsd:complexType name="TrackType">
        <xsd:sequence>
            <xsd:element name="album_name" type="xsd:string"></xsd:element>
            <xsd:element name="artist_name" type="xsd:string"></xsd:element>
            <xsd:element name="title" type="xsd:string"></xsd:element>
            <xsd:element name="play_count" type="xsd:integer"></xsd:element>
            <xsd:element name="rating">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:integer">
                        <xsd:minInclusive value="0"/>
                        <xsd:maxInclusive value="100"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:element>
            <xsd:element name="hash" type="xsd:string"></xsd:element>
            <xsd:element name="kind" type="xsd:string"></xsd:element>
            <xsd:element name="podcast" type="xsd:boolean"></xsd:element>
            <xsd:element name="resource_id" type="xsd:integer"></xsd:element>
            <xsd:element name="created_at" type="xsd:dateTime"></xsd:element>
            <xsd:element name="updated_at" type="xsd:dateTime"></xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="playlist" type="tns:PlaylistType"></xsd:element>
    <xsd:complexType name="PlaylistType">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"></xsd:element>
            <xsd:element name="resource_id" type="xsd:integer"></xsd:element>
            <xsd:element name="created_at" type="xsd:dateTime"></xsd:element>
            <xsd:element name="updated_at" type="xsd:dateTime"></xsd:element>
            <xsd:element name="track_id" type="xsd:integer" minOccurs="0" maxOccurs="unbounded"></
                xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="InfoRegister" type="tns:InfoRegisterType"></xsd:element>
    <xsd:complexType name="InfoRegisterType">
        <xsd:sequence>
            <xsd:element name="username" type="xsd:string"></xsd:element>
            <xsd:element name="catalog_id" type="xsd:integer"></xsd:element>
            <xsd:element name="catalog_type">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                        <xsd:pattern value="Files|iTunes|Amarok"/>
                    </xsd:restriction>
                </xsd:simpleType>
```

```
            </xsd:element>
            <xsd:element name="playlists">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="playlist" type="tns:PlaylistType" minOccurs="0" maxOccurs="
                            unbounded"></xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="tracks">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="track" type="tns:TrackType" minOccurs="0" maxOccurs="unbounded"><
                            /xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="created_at" type="xsd:dateTime"></xsd:element>
            <xsd:element name="updated_at" type="xsd:dateTime"></xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

C.2.2. Procesamiento información musical del usuario

Procesamiento iTunes

```
XMLPropertyListConfiguration xmlplistConf = new XMLPropertyListConfiguration();
xmlplistConf.setFile(fichero_bilioteca); // Se establece el fichero a cargar
xmlplistConf.setEncoding("UTF-8"); // Se establece la codificación de carácteres
xmlplistConf.load(); // Se realiza la carga del fichero
// Se procesa el fichero
Object obj = xmlplistConf.getProperty("Playlists");
if (obj instanceof Collection) {
   // Se procesan los elementos contenidos en el nodo playlists
   Playlist[] playlists = this.extractPlaylists((Collection) obj,xmlplistConf);
   // Se añade la información al objeto library
   if (playlists.length > 0) {
      for (int i = 0; i < playlists.length; i++) {
         library.addPlaylist((Playlist) playlists[i]);
      }
   }
}
```

```
public Playlist[] extractPlaylists(Collection collec, XMLPropertyListConfiguration xmlplistConf) {
   // Se crean tantas playlists como contenga el elemento playlists
   Playlist[] playlists = new Playlist[((Collection) collec).size()];
   // Se extrae la información de cada playlist
   int i = 0;
   for (Iterator iterObj = collec.iterator(); iterObj.hasNext();) {
      HierarchicalConfiguration sub = (HierarchicalConfiguration) iterObj.next();
      playlists[i] = extractPlaylistInfo(xmlplistConf, sub);
      i++;
   }
   return playlists;
}
```

```
public Playlist extractPlaylistInfo(XMLPropertyListConfiguration xmlplistConf, HierarchicalConfiguration
    conf) {

   Playlist playlist = new Playlist(conf.getString("Name"));
   Object itemObj = conf.getProperty("Playlist␣Items");
      if (itemObj instanceof Collection) {
         for (Iterator iterItemObj = ((Collection) itemObj).iterator(); iterItemObj.hasNext();) {
            HierarchicalConfiguration subItems = (HierarchicalConfiguration) iterItemObj.next();
```

```
            Integer trackID = subItems.getInt("Track ID");
            try {
               SubnodeConfiguration track = xmlplistConf.configurationAt("Tracks." + trackID.toString());
               if (isTrackInfo(track, trackID)) {
                  Track aux;
                  try {
                      aux = this.extractTrackInfo(track, trackID);
                      playlist.addTrack(aux);
                  } catch (TunesException e) {
                     auxLogger.warn(null,e);
                  }
               }
            } catch (IllegalArgumentException e) {
                auxLogger.error("ERROR: El key " + "Tracks." + trackID.toString() + " no se corresponde con
                    un único nodo", e);
            }

         }
      }
   return playlist;
}
```

Procesamiento directo de ficheros de audio

```
public Track readTags(String file) throws TunesException {
    try {
        // Biblioteca prioritaria
        MusicMetadataSet src_set = new MyID3().read(new File(file)); // read metadata
        if (src_set != null) {// perhaps no metadata
            Logger.getLogger(FilesParser.class.getName()).debug("file name: " + file + " - MyID3");
            IMusicMetadata metadata = src_set.getSimplified();
            return readMetadata(metadata);
        } else {
            return readTagsOtherLibraries(file);
        }
    } catch (TunesException ex) {
        Logger.getLogger(FilesParser.class.getName()).warn("Probamos con otra librería", ex);
        return readTagsOtherLibraries(file);
    } catch (FileNotFoundException ex) {
        Logger.getLogger(FilesParser.class.getName()).warn(null, ex);
    } catch (IOException ex) {
        Logger.getLogger(FilesParser.class.getName()).warn(null, ex);
    }
    return null;
}
```

```
private Track readMetadata(IMusicMetadata metadata) throws TunesException {
    StringBuffer str = new StringBuffer();
    Track track = new Track();

    String title = metadata.getSongTitle();
    if (validatesNames(title)) {
        str.append(title);
        track.setTitle(title);
    } else {
        throw new TunesException("Track title not valid. Title -> " + title);
    }

    String album = metadata.getAlbum();
    if (validatesNames(album)) {
        str.append(album);
        track.setAlbumName(album);
    } else {
        throw new TunesException("Album name not valid. Album -> " + album);
    }
```

```
    String artist = metadata.getArtist();
    if (validatesNames(artist)) {
        str.append(artist);
        track.setArtistName(artist);
    } else {
        throw new TunesException("Artist name not valid. Artist -> " + artist);
    }
    return track;
}
```

```
public Track readTagsOtherLibraries(String file) {
    try {
        MediaFile mediaFile = new MP3File(new File(file));
        for (Object obj : mediaFile.getTags()) {
            if (obj instanceof ID3V1_0Tag) {
                Logger.getLogger(FilesParser.class.getName()).debug("file name: " + file + " - ID3V1_OTag");
                return readID3V1Tags(obj);
            } else if (obj instanceof ID3V2_3_0Tag) {
                Logger.getLogger(FilesParser.class.getName()).debug("file name: " + file + " - ID3V2_3Tag");
                return readID3V2_3Tags(obj);
            }
        }
         // Se utiliza otra biblioteca para las otras versiones de ID3
        org.farng.mp3.MP3File mp3file = new org.farng.mp3.MP3File(new File(file), false);
        if (mp3file.hasID3v1Tag()) {
            Logger.getLogger(FilesParser.class.getName()).debug("file name: " + file + " - ID3V1_1Tag");
            ID3v1 id3v1 = mp3file.getID3v1Tag();
            return readID3V1_1Tags(id3v1);
        } else if (mp3file.hasID3v2Tag()) {
            Logger.getLogger(FilesParser.class.getName()).debug("file name: " + file + " - ID3V2_2Tag");
            ID3v2_2 id3v2_2 = new ID3v2_2(new RandomAccessFile(file, "r"));
            return readID3V2_2Tags(id3v2_2);
        }
    } catch (IOException ex) {
        Logger.getLogger(FilesParser.class.getName()).warn(null, ex);
    } catch (TagException ex) {
        Logger.getLogger(FilesParser.class.getName()).warn(null, ex);
    } catch (TunesException ex) {
        Logger.getLogger(FilesParser.class.getName()).warn(null, ex);
    } catch (ID3Exception ex) {
        Logger.getLogger(FilesParser.class.getName()).warn(null, ex);
    }
    return null;
}
```

Procesamiento Amarok

```
public LibraryAmarok extractInfo() throws SQLException {
    File libraryFile = getFiletoprocess();
    Connection conn = null;
    try {
        Class.forName("org.sqlite.JDBC");
    } catch (java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException: ");
        System.err.println(e.getMessage());
    }
    try {
        System.out.println("Trying to connect...");
        String path = libraryFile.getPath();
        conn = DriverManager.getConnection("jdbc:sqlite:" + path);
        System.out.println("Connected!");
    } catch (SQLException ex) {
        System.err.print("SQLException: ");
        System.err.println(ex.getMessage());
        ex.printStackTrace();
    }
```

```
    Statement stat = conn.createStatement();
    LibraryAmarok libraryamarok = new LibraryAmarok();

    ResultSet rs = stat.executeQuery("SELECT␣value␣FROM␣admin␣" + "WHERE␣noption␣=␣'Database␣Version'");
    while (rs.next()) {libraryamarok.setDBVersion(rs.getInt("value"));}

    rs = stat.executeQuery("SELECT␣value␣FROM␣admin␣" + "WHERE␣noption␣=␣'Database␣Stats␣Version'");
    while (rs.next()) {libraryamarok.setDBStatsVersion(rs.getInt("value"));}

    rs = stat.executeQuery("SELECT␣value␣FROM␣admin␣" + "WHERE␣noption␣=␣'Database␣Persistent␣Tables␣Version
        '");
    while (rs.next()) {libraryamarok.setDBPersistentTablesVersion(rs.getInt("value"));}

    // Se supone una úncia lista de reproducción
    Playlist biblioteca = new Playlist("BIBLIOTECA");
    rs = stat.executeQuery("SELECT␣t.title,␣al.name␣as␣album,␣ar.name␣as␣artist,␣t.url,␣" +
        "t.filetype,␣st.percentage,␣st.rating,␣st.playcounter,␣st.uniqueid␣" +
        "FROM␣tags␣t,␣album␣al,␣artist␣ar,␣statistics␣st␣" +
        "WHERE␣al.id␣=␣t.album␣AND␣ar.id␣=␣t.artist␣AND␣st.url␣=␣t.url");
    // Con el resultado de la consulta se crea toda la libreria.
    biblioteca.setTracks(extractTrackInfo(rs));
    libraryamarok.addPlaylist(biblioteca);

    conn.close();
    return libraryamarok;
}
```

```
public Vector<Track> extractTrackInfo(ResultSet rs) throws SQLException {
    Vector<Track> tracks = new Vector<Track>();
    while (rs.next()) {
        String title = rs.getString("title");
        String album = rs.getString("album");
        String artist = rs.getString("artist");
        Integer rating = new Integer(rs.getInt("rating"));
        Integer playcounter = new Integer(rs.getInt("playcounter"));
        // Con los datos se crea el track
        Track track = new Track();
        track.setTitle(title);
        track.setAlbumName(album);
        track.setArtistName(artist);
        track.setPlayCount(playcounter);
        track.setRating(rating);
        tracks.add(track);
    }
    rs.close();
    return tracks;
}
```

Bibliografía

[1] Fundación de la Innovación. Bankinter. Web 2.0 el negocio de las redes sociales. 2007.

[2] Ed Ort, Sean Brydon, y Mark Basler. *Mashup Styles, Part 1: Server-Side Mashups* [online]. Mayo 2007. Disponible en: http://java.sun.com/developer/technicalArticles/J2EE/mashup_1/ [última consulta Noviembre de 2007].

[3] Anónimo. *JavaScript: Use a Web Proxy for Cross-Domain XMLHttpRequest Calls* [online]. Disponible en: http://developer.yahoo.com/javascript/howto-proxy.html [última consulta Noviembre de 2007].

[4] Anónimo. *SOAP* [online]. Septiembre 2007. Disponible en: http://rest.blueoxen.net/cgi-bin/wiki.pl?RestInPlainEnglish [última consulta Noviembre de 2007].

[5] Anónimo. *REpresentational State Transfer* [online]. Disponible en: http://en.wikipedia.org/wiki/Representational_State_Transfer [última consulta Ocutbre de 2008].

[6] Anónimo. *REpresentational State Transfer* [online]. Disponible en: http://es.wikipedia.org/wiki/Representational_State_Transfer [última consulta Ocutbre de 2008].

[7] World Wide Web Consortium. *SOAP* [online]. Disponible en: http://www.w3.org/TR/soap/ [última consulta Octubre de 2008].

[8] Anónimo. *SOAP* [online]. Disponible en: http://en.wikipedia.org/wiki/SOAP [última consulta Ocutbre de 2008].

[9] Anónimo. *SOAP* [online]. Disponible en: http://es.wikipedia.org/wiki/SOAP [última consulta Octubre de 2008].

[10] Kenneth Barcklay y John Savage. *Groovy Programming An Introduction For Java Developers*. Morgan Kaufmann, 2007.

[11] Graeme Keith Rocher. *The Definitive Guide To Grails*. Apress, 2006.

[12] Venkat Subramaniam. *The Definitive Guide To Grails*. The Pragmatic Bookshelf, 2008.

[13] Patrick Lenz. *Building Your Own Ruby On Rails Web Applications*. Sitepoint, 2007.

[14] Dave Thomas y David Heinemeier Hansson. *Agile Web Development With Rails, 2ª Ed*. The Pragmatic Programmers, 2005.

[15] Alan Bradburne. *Practical Rails Social Networking Sites*. Apress, 2007.

[16] Eldon Alameda. *Practical Rails Projects*. Apress, 2007.

[17] Obie Fernandez. *The Rails Way*. Addison Wesley, 2008.

[18] Dave Thomas, Chad Fowler, y Andy Hunt. *Programming Ruby, The Pragmatic Programmers' Guide, 2ª Ed*. The Pragmatic Programmers, 2005.

[19] Ben Scofield. *Practical REST On Rails 2 Projects*. Apress, 2008.

[20] Ralf Wirdemann y Thomas Baustert. *Last.fm* [online]. Disponible en: http://www.b-simple.de/download/restful_rails_es.pdf [última consulta Noviembre de 2007].

[21] David Heinemier Hansson. *Rails 1.2: REST admiration, HTTP lovefest, and UTF-8 celebrations* [online]. Enero 2007. Disponible en: http://weblog.rubyonrails.org/2007/1/19/rails-1-2-rest-admiration-http-lovefest-and-utf-8-celebrations [última consulta Ocutbre de 2007].

[22] Anónimo. *Iniciándonos con REST en RAILS* [online]. Enero 2007. Disponible en: http://www.cduv.org/2007/01/20/inicindonos-con-rest-en-rails/ [última consulta Octubre de 2007].

[23] Peepcode. *Peepcode REST cheatsheet* [online]. 2007. Disponible en: http://peepcode.com/system/uploads/REST-cheatsheet.pdf [última consulta Noviembre de 2007].

[24] John Nunemaker. *Scrobbler: Last.fm For Ruby* [online]. Mayo 2007. Disponible en: http://www.railstips.org/2007/5/11/scrobbler-last-fm-for-ruby [última consulta Noviembre de 2007].

[25] World Wide Web Consortium. *WSDL* [online]. Disponible en: http://www.w3.org/TR/wsdl/ [última consulta Octubre de 2008].

[26] Sean Lynch. *Rails 2.0 and Scaffolding Step by Step* [online]. 2007. Disponible en: http://fairleads.blogspot.com/2007/12/rails-20-and-scaffolding-step-by-step.html [última consulta Septiembre de 2008].

[27] Anónimo. *Restful Authentication with all the bells and whistles (original)* [online]. Noviembre 2007. Disponible en: http://railsforum.com/viewtopic.php?pid=74245#p74245 [última consulta Septiembre 2008].

[28] Daniel Fischer. *How to use GMail as your mail server for Rails* [online]. Enero 2008. Disponible en: http://www.danielfischer.com/2008/01/09/how-to-use-gmail-as-your-mail-server-for-rails/ [última consulta Septiembre 2008].

[29] Jaime Iniesta. *Envio de correos a través de GMail con Rails* [online]. Junio 2008. Disponible en: http://www.jaimeiniesta.com/2008/07/16/envio-de-correos-a-traves-de-gmail-con-rails/ [última consulta Septiembre 2008].

[30] W. Clay Richardson, Donald Avondolio, Joe Vitale, Scot Chrager, y Jeff Scanlon. *Java 2 v5.0*. Anaya Multimedia, 2005.

[31] Ed Ort y Bhakti Mehta. *Java Architecture for XML Binding (JAXB)* [online]. Marzo 2003. Disponible en: http://java.sun.com/developer/technicalArticles/WebServices/jaxb/ [última consulta Noviembre de 2007].

[32] Amy Hoy. *RJS desmitified with pretty colors* [online]. Ocutbre 2006. Disponible en: http://slash7.com/articles/2006/10/8/rjs-demistified-with-pretty-colors [última consulta Septiembre 2008].

[33] Amy Hoy y CM Harrington. *RJS Cheat Sheet* [online]. Octubre 2006. Disponible en: http://localtype.org/rjs/rjs_cheat_sheet.pdf [última consulta Septiembre 2008].

[34] Gregg. *Ruby On Rails Rake tutorial* [online]. Junio 2007. Disponible en: http://www.railsenvy.com/2007/6/11/ruby-on-rails-rake-tutorial [última consulta Septiembre 2008].

[35] Anónimo. *Rake* [online]. Septiembre 2008. Disponible en: http://en.wikipedia.org/wiki/Rake_(software) [última consulta Septiembre 2008].

[36] Tammer Saleh. *BDD With Shoulda* [online]. 2008. Disponible en: http://tammersaleh.com/system/assets/bdd_with_shoulda.pdf [última consulta Octubre de 2008].