Desarrollo de una aplicación web distribuida y escalable para el descubrimiento de música a través de un juego

Jorge Madrid Portillo

Septiembre 2014

A mis abuelos, Arturo, Carmela, Nicolás y María.

Abstract

The arrival of new technologies in the market often raises new problems to be solved. We are nowadays living a radical shift of paradigm on the web world with the coming of new gadgets and electronic devices such as smartphones and tablets. The traditional web systems are mostly becoming obsolete and hardly adaptable to these surrounding changes while classic software development models are not offering any help with this task.

By applying a modern approach to web applications, my partner Carlos Gómez Muñoz and me, have been able to develop an application based on a game that allows the user to discover new music and explore new musical genres. In order to achieve this it has been necessary to rethink the conceptual structure of one the most famous software patterns, the MVC, reorganizing responsibilities with the objective to satisfy the scalability and availability requirements of the system.

Furthermore we have had to explore new technologies and systems such as Cloud Computing or new programming languages and frameworks to satisfy in a simple and economical manner the system necessities. The result has been the design and implementation of a system based on a modern and contemporary technology stack able to respond to the possible changes regarding scalability and availability.

Keywords: REST; JSON; NodeJS; OAuth; MongoDB; Nginx; escalability; avalability.

Resumen

Con la llegada de nuevas tecnologías al mercado a menudo surgen nuevos problemas a resolver. En el caso de la web, actualmente se esta viviendo un cambio de paradigma radical con la llegada de nuevos dispositivos como tablets, smartphones...etc. Los sistemas tradicionales han quedado en su mayoría obsoletos siendo dificilmente adaptables a estos cambios mientras que los modelos clásicos de desarrollo no facilitan dicha tarea.

A través de un enfoque moderno a las aplicaciones web, mi compañero Carlos Gómez Muñoz y yo, hemos sido capaces de desarrollar una aplicación basada en un juego que permite al usuario descubrir nueva música y explorar nuevos estilos musicales. Para ello ha sido necesario replantear la estructura conceptual del famoso patrón MVC con el objetivo de reorganizar las responsabilidades y conseguir satisfacer los requisitos de escalabilidad y disponibilidad del sistema.

De igual manera hemos tenido que explorar las nuevas tecnologías y sistemas como los servicios de Cloud Computing o nuevos lenguajes de programación y frameworks para satisfacer las necesidades del sistema de una forma sencilla y económica. El resultado ha sido el diseño e implementación de un sistema basado en un "stack"de tecnologías contemporáneas capaz de responder ante los posibles cambios de escalabilidad y disponibilidad.

Palabras clave: REST; JSON; NodeJS; OAuth; MongoDB; Nginx; escalabilidad; disponibilidad.

Agradecimientos

Tras cuatro años de carrera puedo ver en perspectiva lo que siginifica alcanzar una meta y los problemas a los que me he enfrentado para conseguirlo. Todo este trabajo no hubiera sido posible sin la ayuda de muchas personas a las cuales me dispongo a mostrar todo mi agradecimiento.

En primer lugar y sobretodo a mi madre por su comprensión y constante apoyo, nada de lo que he conseguido en esta vida hubiera sido posible sin ella y sin su inagotable paciencia. Por su puesto a mi padre por su ayuda a lo largo de toda la carrera y sus consejos que aunque no lo parezca siempre son útiles. Gracias también a mi hermano Raúl por ayudarme en los últimos y duros momentos y por su curiosidad, espero que pueda servirte de insipiración para futuros trabajos.

Agradecer también a mi inestimable amigo y compañero de viaje Carlos por amenizar nuestros "sprints"de programación y por compartir tantos momentos, algunos de tensión, tanto en el proyecto como durante toda la carrera. Dar gracias también a mis compañeros de CEURA y Telecor y en especial a mi compañero Miguel por ser una fuente inagotable de conocimientos que afortunadamente he tenido el privilegio de recibir en muchas ocasiones. Agradecimientos tambień a Nacho y Alfonso, socios de Planum y grandes amigos, por todo el apoyo prestado.

A mi director de proyecto, profesor y compañero, Rodrigo por su fé en el proyecto, su ayuda y correcciones. Agradecer también a mi profesor de Ingeniería del Software, Raúl, por enseñarme tanto en tan poco tiempo y por demostrar que la programación dirigida por pruebas es posible.

Gracias a Ana, amiga y compañera de trabajo por su interés, apoyo en los momentos más duros y ayuda en forma de azúcar a la que solemos llamar chucherías. Agradecer por último a mi reciente compañero y jefe de trabajo, Ricardo por su apoyo y compresión en Telecor.

Sirva este texo para agradecer también a aquellas personas que en mayor o menor medida han contribuido a que pueda llegar hasta donde estoy ahora, amigos, compañeros, familiares... Gracias a todos.

Índice general

Índice de figuras

Capítulo 1

Introducción

Internet ha revolucionado la industria musical en los últimos 15 años. Desde Napster e iTunes, pasando por Spotify o incluso YouTube, la forma en la que las personas descubren y comparten música ha cambiado radicalmente. Tener al alcance de un click la mayor parte de la música existente permite a los usuarios descubrir mucha más música que antes.

Los sistemas de recomendación que ofrecen estos servicios a sus usuarios permiten descubrir nuevos artistas y canciones de forma automática y bastante eficaz. Éstos devuelven en sus resultados música similar a la reproducida anteriormente por el usuario (en función de clasificaciones por estilos ej. rock, pop, clásica) o bien música reproducida por usuarios similares (a partir de datos análogos al anterior caso). Estas herramientas son realmente útiles, pero en cierta medida se pierde el "factor humano". Por otra parte, hace no tanto tiempo, las mejores formas (e incluso la únicas) de conocer nueva música eran a través de la radio o por simple boca a boca; algún amigo nos recomendaba una nueva canción que él pensaba que nos podría gustar.

El boca a boca, en cierta medida, se ha mantenido e incluso mejorado gracias al uso de las redes sociales, pero los usuarios a veces no tienen tiempo o interés en compartir y/o descubrir música de forma activa aún cuando posiblemente aceptarían de buen grado nueva música que les gustara.

1.1. Normas del juego

El objetivo del juego es dar a conocer a un grupo cerrado de personas nueva música de forma que la canción compartida sea la que más gusta, y a la vez conocer y **puntuar la música** que estas personas te dan a conocer a ti.
El juego Planum está pensado para a partir de 4 jugadores, sin un límite concreto, pero se recomienda entre 4 y 7 jugadores (tantos jugadores como días de la semana).

Cada jugador tendrá preparada en cola al menos una canción, y aleatoriamente, cada día de la semana se publicará una canción de un jugador de forma anónima. Los jugadores tendrán que puntuarla mediante las etiquetas, tanto a esta canción como a las del resto de la semana. Tras votar todas las canciones, se proclamará un **ganador** de lo que llamamos ronda, y se comenzará una nueva ronda. Se jugarán tantas rondas como se configure al comienzo.

Capítulo 2

Estado del arte

La mayor parte de las aplicaciones web de hoy en día siguen un esquema básico de modelo cliente servidor utilizando los protocolos estándares de la web. Puesto que dichas aplicaciones no tienen requisitos específicos de disponibilidad, escalabilidad, rendimiento... lo normal hasta ahora ha sido realizarlas usando los paradigmas clásicos de programación y usando las mismas pilas de tecnologías LAMP por ejemplo). Sin embargo, debido a los crecientes cambios en el mundo tecnológico se ha vuelto imprescindible ampliar cambiar la forma en la que se diseñan e implementan los sistemas web.

En este capítulo se presenta el estado en el que se encuentran las tecnologías usadas para llevar a cabo el proyecto. Esta divido en cuatro secciones, tecnologías, lenguajes de programación y frameworks, sistemas y bases de datos.

2.1. Tecnologías

2.1.1. SOAP

SOAP (*Simple Object Access Protocol*) es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse a través de intercambio de datos en formato XML.

El protocolo SOAP es uno de los protocolos más utilizados en los servicios Web. Puesto que el protocolo debe funcionar a través de la infraestructura establecida para las comunicaciones en la web, SOAP se apoya en el protocolo

de aplicación HTTP entre otros, con el fin de facilitar las comunicaciones entre los distintos actores de la web. Las características principales son las siguientes:

- **Extensibilidad**(seguridad y WS-routing son extensiones aplicadas en los desarrollos con SOAP).
- **Neutralidad** (SOAP puede ser utilizado sobre cualquier protocolo de transporte como HTTP, SMTP, TCP o JMS).
- **Independencia** (SOAP permite cualquier modelo de programación).

Uno de los grandes problemas del protocolo SOAP es la **dificultad de implementación** dentro de un sistema. Es por ello que surgen nuevas tecnologías que facilitan la implementación de sistemas que cumplan el paradigma de arquitectura cliente servidor y permita las ventajas de un protocolo extensible y que permita la comunicación de distintos actores.

2.1.2. REST

El 'protocolo' REST (*Representational State Transfer*) es una técnica de arquitectura de software para sistemas distribuidos en la web basada en una tesis doctoral por Roy Fielding en el año 2000. La principal ventaja de REST frente a protocolos alternativos como SOAP es su facilidad de uso. Las aplicaciones desarrolladas con REST son más sencillas, y aprovechan las características intrínsecas de HTTP.

A diferencia de los protocolos estilo RPC (*Remote Procedural Call*), REST se orienta en torno a los recursos de una aplicación, aprovechando las operaciones GET, POST, DELETE, PUT, por ejemplo.

- La operación **getBankAccount** sería equivalente a una request tipo GET a **http://domain.com/bankAccount**
- Del mismo modo la operación **deleteBankAccount** sería equivalente a una request tipo DELETE a **http://domain.com/bankAccount**, y así sucesivamente con el resto de operaciones y recursos.

Una de las asociaciones más comunes pero erróneas consiste en identificar los cuatro métodos HTTP **(GET, POST, PUT y DELETE)** con las operaciones básicas CRUD (*Create, Read, Update y Delete*) respectivamente.

Mientras que las operaciones GET y DELETE si pueden asociarse a Create y Delete, las operaciones POST Y PUT mantienen un debate abierto respecto a como deberían actuar con los recursos del servidor.

Esto último es debido a que POST es el único de dichos verbos que no es idempotente, es decir, varias llamadas a la misma URL pueden producir distintos resultados. Teniendo eso en cuenta, una buena forma de utilizar PUT y POST podría ser:

- **PUT** se debe usar para crear o actualizar recursos cuando se conoce la URL completa (con identificador) del recurso: PUT /bankAccount/123.
- **POST** se debe usar para crear o actualizar recursos cuando no se conoce la URL completa. En este caso varias llamadas a la misma URL podrían resultar en la creación de varios recursos.

Es posible encontrarse con operaciones que no estén cubiertas por las operaciones estándar HTTP. En ese caso es posible utilizar el verbo POST como última opción para realizar operaciones concretas sobre un recurso pasando el nombre de la operación deseada mediante parámetros en la URL del siguiente modo: POST http://domain.com/bankAccount?operation=freeze.

Las aplicaciones REST pueden realizar el intercambio de datos en varios idiomas, los más comunes suelen ser XML y JSON (Javascript object notation). Para mi aplicación he escogido JSON debido a su simpleza y fácil traducción a objetos en el lenguaje javascript.

2.1.3. OAuth

OAuth(*Open Authorization*) es un protoclo abierto, propuesto por Blaine Cook y Chris Messina, que permite autorización segura de una API de modo estándar para aplicaciones de escritorio, móviles y web.

Mediante el protocolo OAuth una aplicación es capaz de autenticarse frente a un **servidor de autenticación** concreto y recibir las credenciales necesarias para hablar con el API al que hay obtenido autorización.

En mi caso he usado la tecnología de OAuth, la cual garantiza el acceso a las API's a través de **tokens de sesión**. Dichos tokens tienen un tiempo de vida determinado y pueden ser renovados haciendo más peticiones contra el servidor de autenticación. En cualquier caso, el protocolo será explicado con más detalle en futuros apartados.

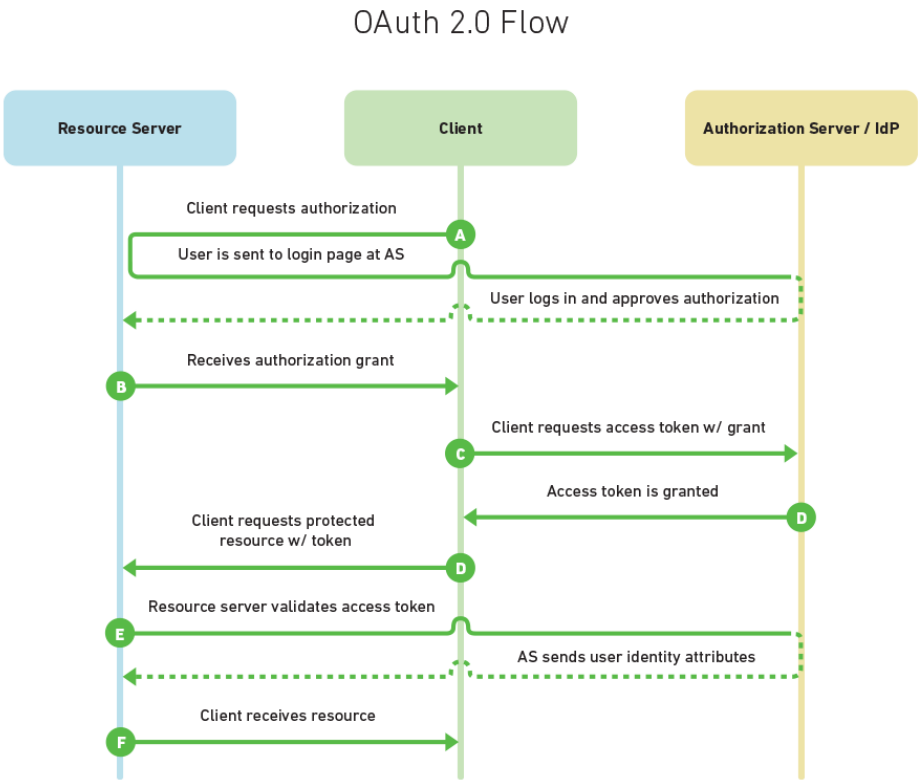

Figura 2.1: Caso de uso típico para autenticación OAuth2

2.2. Frameworks y lenguajes

En este apartado se describen brevemente los frameworks y lenguajes de programación utilizados en este proyecto, así como los los más utilizados en las aplicaciones web desplegadas actualmente.

2.2.1. PHP

PHP (*PHP Hypertext Processor*) es uno de los lenguajes más utilizados en el mundo de la tecnología web junto con muchos frameworks desarrollados por encima como Cake.php, Codeginiter...

Esto es debido a que resulta muy **fácil y cómodo encontrar hosting** gratuito o muy barato que te permita alojar de forma muy sencilla aplicaciones PHP aprovechando el clásico stack tecnológico LAMP (*Linux Apache MySql y PHP*). Además PHP a demostrado ser escalable en aplicaciones a muy gran escala como Facebook o Tuenti entre otras.

A diferencia de node.js, PHP es un lenguaje orientado plenamente a la Web y permite mezclar código **PHP y HTML** en un mismo fichero sin necesidad de módulos añadidos. PHP es hoy en día junto con Java uno de los lenguajes más utilizados en la Web para la parte servidor, sin embargo, el desarrollo de aplicaciones distribuidas para múltiples dispositivos ha hecho que PHP se quede atrás, dado que no está igual de capacitado para la implementación de API's REST y no soporta el mismo número de conexiones concurrentes que el modelo orientado a eventos.

2.2.2. NodeJS

Node.js es una plataforma construida encima del entorno de ejecución de **Javascript de Google Chrome**. Permite construir aplicaciones en red en javascript para la parte servidor de una manera rápida y escalable, perfecto para aplicaciones distribuidas que requieren un intercambio intensivo de datos.

La principal ventaja de node.js frente a los modelos clásicos de programacón en la web es el diseño de la capa de **entrada y salida (I/O) orientada a eventos**. De esta forma todos los accesos a los recursos del sistema (red, disco...) son asíncronos de forma nativa por lo que ya no es necesario implementar complejos sistemas de hilos para las operaciones costosas. Además existen numerosos módulos para la plataforma de node.js como express.js,

connect.js..etc que facilitan mucho el desarrollo de aplicaciones distribuidas a través de la red.

Además de la alta velocidad de ejecución de Javascript, la verdadera magia detrás de Node.js es algo que se llama Bucle de Eventos (**Event Loop**). El enfoque tradicional para generar código asíncrono es complejo y crea un espacio en memoria no trivial para un gran número de clientes ya que cada cliente crea un nuevo hilo en el sistema operativo. Para evitar esta ineficiencia, así como la dificultad conocida de las aplicaciones basadas en hilos, Node.js mantiene un bucle de eventos (Event Loop) que gestiona todas las operaciones asíncronas. Cuando una aplicación Node.js necesita realizar una operación de bloqueo (operaciones I/O como trabajo con archivos ... etc) envía una tarea asíncrona al bucle de eventos, junto con un callback, y luego continúa.

Como indica la figura, el bucle de eventos realiza un seguimiento de la operación asincrónica, y ejecuta el **callback** cuando finalice, retornando los resultados a la aplicación. Esto le permite gestionar un gran número de operaciones, tales como conexiones de clientes o cálculos, dejando que el event loop maneje eficientemente el grupo de subprocesos. Por supuesto, dejando esta responsabilidad al bucle de eventos hace la vida mucho mas fácil para los desarrolladores, quienes pueden enfocarse en la funcionalidad de la aplicación.

Si se combina el stack de *node.js + REST (json) + base de datos con formato json*, resulta muy sencillo trabajar con los datos ya que no es necesario realizar en ningún momento conversión alguna, puesto que javascript es capaz de entender los objetos json de forma nativa.

2.2.3. Java, JSP

Java ó JSP (*Java Server Pages*) es una de las tecnologías **más usadas en el mundo de la empresa** para crear aplicaciones web escalables y seguras. La plataforma Java ha demostrado ser una de las más fiables y seguras para aquellos sistemas que requieran una gran cantidad de usuarios. El concepto de los servlets de Java es similar al de PHP en el sentido que se crea un nuevo hilo o proceso por cada nueva petición. La principal ventaja de JSP frente a otros lenguajes es que el lenguaje Java es un lenguaje de **propósito general** que excede el mundo web y que es apto para crear clases que manejen lógica de negocio y acceso a datos de una manera efectiva. Esto permite separar en niveles las aplicaciones web, dejando la parte encargada de generar el

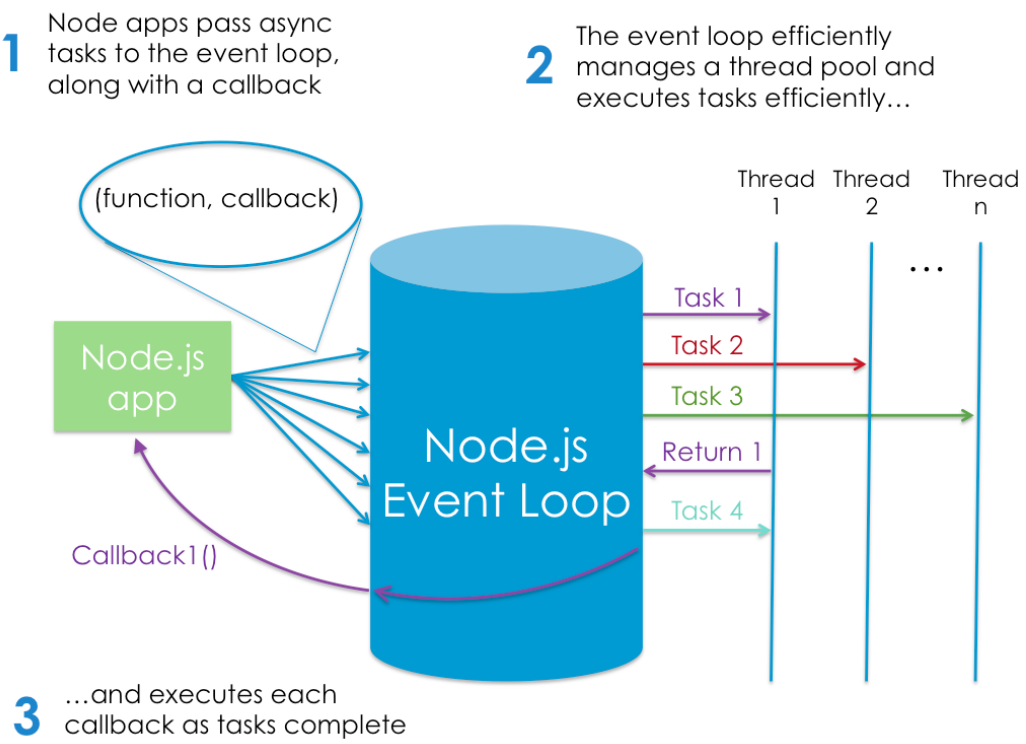

Figura 2.2: Bucle de eventos asíncrono de node.js

documento HTML en el archivo JSP.

El problema que tiene la tecnología Java frente a otras es que necesita **muchos recursos** de sistema para ser desplegada correctamente en un entorno con Apache y Tomcat o similar, es por eso que muchos desarrolladores tienden a acercarse a tecnologías como PHP o Python.

2.3. Despliegue de aplicaciones

El mundo de la implantación de aplicaciones a cambiado muy rápidamente con la llegada de tecnologías como la computación en la nube. Es por eso que he analizado varias alternativas con respecto al hosting tradicional.

2.3.1. Servidor Compartido

Este tipo de hosting es un servicio que provee a los usuarios de Internet un sistema para poder almacenar información, imágenes, vídeo, o cualquier contenido accesible vía web. Normalmente estos servicios **restringen** al usuario de obtener control total de la máquina que se está usando, ya que esa máquina probablemente este siendo usada por varios usuarios para desplegar sus aplicaciones.

Este tipo de hosting suele ser útil para el despliegue de **aplicaciones sencillas** en PHP con MySql o similar.

2.3.2. Servidores dedicados

Los servidores dedicados son máquinas virtuales o no, con las que el usuario puede realizar cualquier actividad o uso necesario para el despliegue de sus aplicaciones. A diferencia de lo que ocurre con el alojamiento compartido, en donde los recursos de la máquina son compartidos entre un número indeterminado de clientes, en el caso de los servidores dedicados generalmente es un solo cliente el que dispone de **todos los recursos** de la máquina para los fines por los cuales haya contratado el servicio.

Las principales ventajas de un servidor dedicado frente un servidor compartido son las siguientes:

- Disponibilidad de los recursos completos de la máquina.
- Mayor capacidad de configuración del sistema.

- Mayor control sobre las aplicaciones.

Sin embargo existe una desventaja muy importante en este tipo de servidores: **el coste del servicio**, que suele ser mucho mayor al de los servidores compartidos. Estos servidores suelen utilizarse para desplegar aplicaciones que requieran muchos recursos como Java.

2.3.3. Hosting en la nube

El Hosting en la nube es una solución que combina lo mejor del hosting compartido y de los servidores dedicados. Las grandes empresas como **Amazon o Google** aprovechan su infraestructura de computación en la nube para ofrecer servicios elásticos de hosting y aplicaciones a otras empresas o particulares. De este modo es posible obtener servidores dedicados por completo de forma barata y pagar tan solo aquellos recursos que estés usando en ese momento.

Además, si en el caso que se quiera ampliar los recursos de las máquinas basta con **agregar nuevas máquinas** o solicitar máquinas más potentes a un coste razonable.

2.3.4. Iaas

En español Infraestructura como Servicio. Modelo de distribución de infraestructura de computación como un servicio, normalmente mediante una plataforma de virtualización. En vez de adquirir servidores, espacio en un centro de datos o equipamiento de redes, los clientes compran todos estos recursos a un proveedor de servicios externo. Una diferencia fundamental con el hosting virtual es que el provisionamiento de estos servicios se hacen de manera integral a través de la web.

Amazon

Amazon Elastic Compute Cloud (Amazon EC2) es un servicio web que proporciona capacidad informática con tamaño modificable en la nube. Está diseñado para facilitar a los desarrolladores recursos informáticos escalables basados en web.

Google Cloud Computing

Google Cloud Platform permite que desarrolladores puedan crear, probar e implementar aplicaciones en la infraestructura escalable de Google. Es posible escoger entre los servicios de computación, almacenamiento y aplicaciones para soluciones web, para móviles y backend.

2.4. Bases de datos

Hoy en día el mundo de las bases de datos se encuentra divido en dos paradigmas, las bases de datos **Sql** y **NoSql**.

2.4.1. Sql

Las bases de datos SQL utilizan el álgebra y el cálculo relacional con el objetivo de realizar consultas para obtener o modificar información de la base de datos. La manera de realizar dichas consultas es a través del lenguaje de consultas estructurado o SQL(*Structured Query Language*). El desarrollo de SQL se atribuye a IBM después de haber analizado las ideas de Codd en 1977.

Hoy en día las bases de datos relacionales SQL son las más usadas dada su **fiabilidad y rendimiento**. Existen varias implementaciones de varias empresas, por ejemplo, OracleSQL, MySQL, MariaSQL...etc. Las principales características de SQL, como se puede observar en la figura 2.4 son:

- Normalización de los datos.
- Integridad de los datos a través de restricciones en las tablas.
- Atomicidad de los datos.
- Control de transacciones.
- Creación de disparadores o triggers.
- Operaciones de relación de tablas (Join).

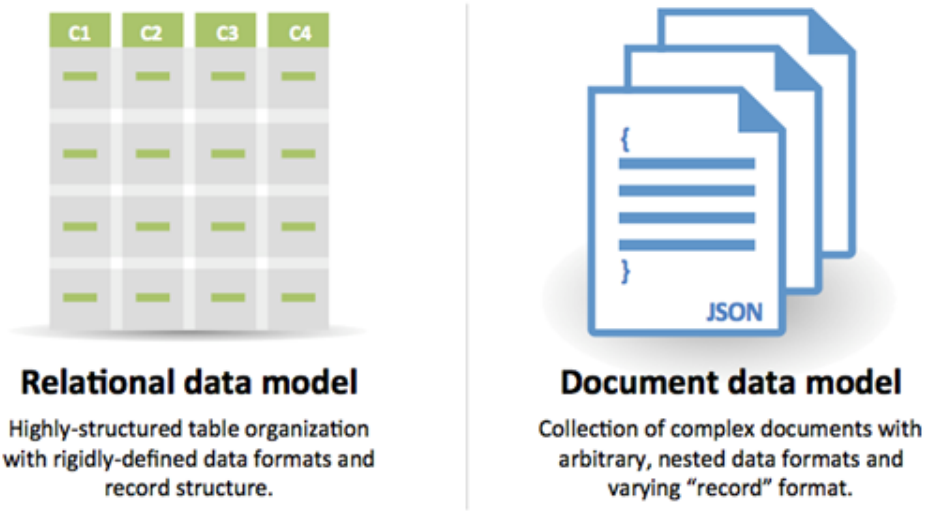

Figura 2.3: Esquéma de datos sql vs base de datos orientada a documentos

2.4.2. NoSql

El termino NoSql hace referencia a un conjunto de sistemas de gestión de bases de datos que se diferencian del modelo tradicional de sistemas de gestión de bases de datos relacionales (RDBMS) es aspectos importantes como por ejemplo:

- No se utiliza SQL como lenguaje de consultas.
- No soportan operaciones de relación (Joins).
- No garantizan completamente la definición de ACID.
- No se organizan mediante tablas como las bases de datos relacionales.

Existen varios tipos de bases de datos relacionales, las más comunes por forma de almacenar los datos son las siguientes: **clave-valor, BigTable, orientadas a documentos y orientadas a grafos**. Los impulsores de este tipo de tecnologías han sido empresas grandes como Google que se han visto en la necesidad de almacenar grandes cantidades de datos sin que importe demasiado la estructura de estos, sino el rendimiento de consulta y modificación, o la escalabilidad.

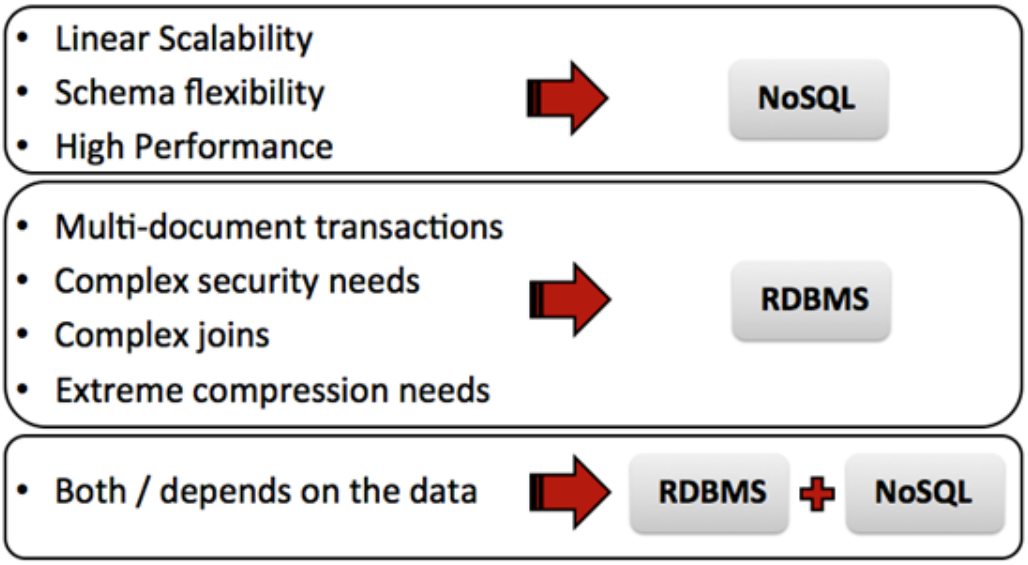

Figura 2.4: Diferencia de características entre sql y nosql

Las principales ventajas de este tipo de bases de datos son las siguientes:

- Mayor escalabilidad horizontal.
- Pueden manejar enormes cantidades de datos.
- Se ejecutan en clusters de máquinas baratas.
- No generan cuellos de botella.

2.4.3. Tecnologías utilizadas

Para poder satisfacer los requisitos de la aplicación, en el desarrollo de esta aplicación se han utilizado las tecnologías siguientes de todas las mencionadas anteriormente:

- REST
- NodeJS
- OAuth2
- Amazon Web Services

Capítulo 3

Diseño

En este capítulo se intenta analizar en profundidad el problema que se pretende resolver, intentando abarcar la mayor cantidad de información posible y estudiando los principales aspectos que habrían de cubrirse en el desarrollo del proyecto. En este punto deberían surgir las características principales del sistema a desarrollar, por lo que se establecen también los casos de uso que deberá poder efectuar el sistema resultado del proyecto para cumplir los requisitos que se establezcan.

3.1. Descripción del problema

El objetivo principal es construir una aplicación web escalable horizontalmente, rápida y que no consuma demasiados recursos del sistema, y que además permita la conexión de múltiples dispositivos como navegadores, móviles o tablets para el intercambio de datos. El problema se podría dividir en varios apartados, diseño de la arquitectura del sistema para satisfacer las necesidades descritas previamente, implementación de dicho sistema, despliegue de las aplicaciones y pruebas.

La aplicación permite a los usuarios disfrutar de la experiencia de un juego mientras que conocen nueva música a través de los diferentes usuarios. Los usuarios se registran en la aplicación y pueden añadir canciones a su cola personal, de forma que a medida que avanza el juego estas se van publicando en las rondas para que otros jugadores puedan disfrutarlas y votarlas.

3.2. Requisitos

La fase de análisis incluye el estudio y la definición de los requisitos del sistema. Los requisitos efinen en detalle los servicios que el sistema debería poder ofrecer, identificando las restricciones y carácterísticas de éste.

3.2.1. Requisitos funcionales

RF1

El sistema debe permitir el registro de usuarios en la base de datos.

RF2

El sistema debe permitir que los clientes obtengan los datos del usuario autenticado.

RF3

El sistema debe garantizar que un cliente obtiene correctamente los datos de los usuarios del juego en curso.

RF4

El sistema debe proporcionar a los clientes los datos del juego en curso

RF5

Los clientes deben poder crear nuevos juegos a través de la aplicación en el sistema.

RF6

El sistema debe otorgar acceso a los datos de las publicaciones y las rondas a los clientes.

RF7

El sistema debe proporcionar acceso a los datos de una publicación en particular.

RF8

Los clientes podrán obtener información sobre los votos o comentarios de una publicación.

RF9

Los clientes tendrán acceso a las etiquetas disponibles del sistema.

RF10

El sistema debe permitir el acceso a la cola de canciones del usuario.

RF11

Los clientes podrán publicar nuevas canciones a su cola.

RF12

Los clientes podrán borrar canciones de la cola.

RF13

El sistema será capaz de buscar canciones y vídeos en la API de youtube.

RF14

Los clientes podrán publicar votos y comentarios en las publicaciones.

RF15

El sistema deberá realizar búsqueda de artistas y canciones en el API de freebase

RF16

El sistema deberá realizar búsqueda de artistas y canciones en el API lastfm

RF17

El sistema debe ser capaz de avanzar el juego diariamente.

RF18

El cliente debe ser capaz de consultar sus datos de perfil y los de los jugadores del juego en curso.

RF19

Los datos del sistema deben poder ser administrados a través de forma privada. Se podrán crear nuevos usuarios, ver información acerca de estos, crear nuevas etiquetas o ver las que hay disponibles, obtener información acerca de los juegos y de las canciones publicadas.

3.2.2. Requisitos no funcionales

Los requisitos no funcionales son requisitos que especifican criterios que pueden usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos, ya que éstos corresponden a los requisitos funcionales. Por tanto, se refieren a todos los requisitos que no describen información a guardar, ni funciones a realizar. Los requisitos no funcionales son **rendimiento y alta disponibilidad** y para ello se va a diseñar un sistema con las siguientes características:

Escalabilidad

La escalabilidad es la capacidad del sistema informático de cambiar su tamaño o configuración para adaptarse a las circunstancias cambiantes sin llegar a deteriorar la calidad del sistema o agotar los recursos. Existen dos tipos de escalabilidad:

- **Escalabilidad vertical**: Se produce cuando al añadir más recursos a un nodo concreto del sistema, la calidad de este mejora en conjunto. Ejemplos de escalabilidad vertical son por ejemplo: añadir más memoria RAM, más disco duro, mayor capacidad de procesamiento...etc.
- **Escalabilidad horizontal**: Un sistema escala horizontalmente cuando al agregar más nodos al sistema, el rendimiento de este mejora en

conjunto también. La escalabilidad horizontal se puede conseguir introduciendo balanceo de carga dentro de un sistema.

En el caso de nuestro sistema la escalabilidad es un requisito no funcional necesario. Las aplicaciones web de hoy en día deben responder ante una variedad creciente de posibles clientes, como tables, móviles, navegadores...etc, lo cual supone mayor número de usuarios. Con la llegada de la computación en la nube es menos costoso crear nuevas instancias o nodos en el sistema que asignar mas recursos a las máquinas disponibles, por lo que es más razonable preparar el sistema para que sea escalable horizontalmente. Además la escalabilidad horizontal permite capacidad de acción rápida si el sistema sufre una llegada masiva de nuevos usuarios, ya que si bien preparado, suele ser tan sencillo como añadir nuevos nodos al cluster.

Rendimiento

El rendimiento es un requisito intrínsecamente **relacionado con la escalabilidad**, puesto que la escalabilidad es la previsión a futuro de una mayor necesidad de rendimiento, con el fin de que la medida óptima de rendimiento no disminuya. En el caso de nuestra aplicación el rendimiento no es un requisito imprescindible, si no más bien una necesidad que tiene que ver con cualquier aplicación web moderna.

Disponibilidad

La disponibilidad es un requisito que garantiza que un sistema permanezca en un continuo grado de operación durante un periodo de tiempo concreto, es decir la capacidad de los usuarios de utilizar el sistema con normalidad a pesar de la carga a la que el servidor pueda estar sometido. Consiste en evitar a toda costa la no disponibilidad del sistema. Esta no disponibilidad puede estar producida por varios motivos. Uno de los motivos puede ser el fallo del motor de base de datos, o por ejemplo la caída de uno de los servidores de la API. Es muy difícil prevenir las caídas de servidores por lo que el mejor mecanismo de actuación es desplegar clusteres de servidores, ya sean de base de datos o de aplicaciones, con replicas o nodos funcionando concurrentemente. De ese modo, si uno de los nodos o replicas falla, otro puede tomar el control o mantener la calidad del servicio.

En el caso de mongoDB lo que se ha utilizado es una solución de replicación que consiste en montar es un 'set de replicas' que puedan funcionar

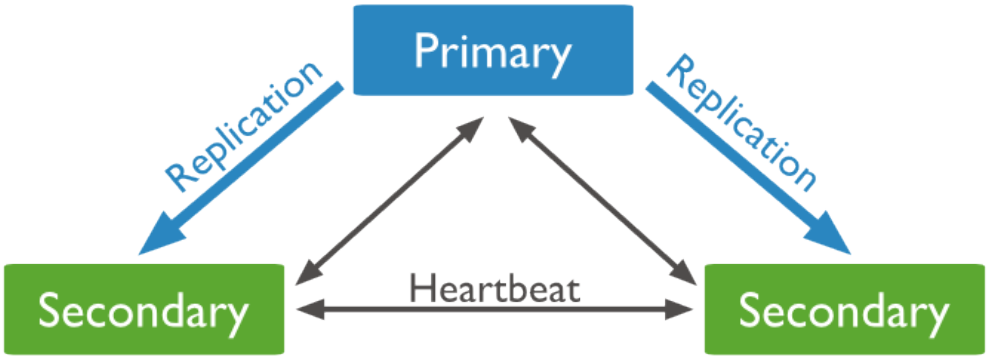

Figura 3.1: Esquema de réplicas de mongoDB

con un paradigma de maestro esclavo, y se comuniquen entre ellas a través de sockets TCP para comprobar si siguen vivas. La implementación de dicho 'set' se verá en el capítulo de implementación.

Para el cluster de APIs REST se ha utilizado un servidor proxy inverso que realiza balanceo de carga entre los servidores API, de forma que si alguno de estos servidores no responde el proxy mandará la petición al siguiente y así sucesivamente.

3.3. Casos de Uso

Los casos de uso representan las diferentes posibles interacciones que puede existir entre los diferentes actores dentro de un sistema. Los actores de un sistema son entidades externas al sistema que pueden interactuar con él. Los actores pueden ser humanos u objetos (servidores, aplicaciones, servicios...). Los actores que interactúan con nuestra aplicación son los siguientes:

- Usuario del sistema, ya sea a través del navegador u otro tipo de dispositivo.
- Cliente del servidor.
- Api de Youtube.
- Api de la ontología de freebase.
- Api REST de lastfm.

CU1: Registro del usuario

El usuario realiza un registro contra el servidor de autenticación a través del navegador u otro dispositivo.
Actores Implicados: Cliente, Sistema.
Precondiciones: El sistema debe estar correctamente funcionando.
Secuencia de acciones:

1. El cliente se descarga la web de los ficheros estáticos del servidor.
2. El cliente envía el formulario de solicitud de registro.
3. El servidor comprueba que no exista el usuario y realiza el registro.

Postcondiciones: Ninguna.

CU2: Login del usuario

El usuario se autentica correctamente en la aplicación
Actores Implicados: Cliente, Sistema.
Precondiciones: El sistema debe estar correctamente funcionando y el usuario debe existir en el sistema.
Secuencia de acciones:

1. El cliente se descarga la web de los ficheros estáticos del servidor.
2. El cliente envía el formulario de autenticación al servidor.
3. El servidor autentica al usuario.
4. El cliente web solicita al servidor los datos de los juegos disponibles.

Postcondiciones: Ninguna.

CU3: Elección de partida

El usuario escoge una partida tras haberse autenticado.
Actores Implicados: Cliente, Sistema.
Precondiciones: El sistema debe estar correctamente funcionando y el usuario autenticado en el cliente
Secuencia de acciones:

1. El cliente se descarga la web de los ficheros estáticos del servidor.
2. El cliente se autentica correctamente y es presentado con la pantalla de selección de juegos.
3. El cliente escoge una partida.
4. El servidor registra la elección de partida.

Postcondiciones: Ninguna.

CU4: Creación de partida

El usuario crea una partida tras haberse autenticado.
Actores Implicados: Usuario, Sistema.
Precondiciones: El sistema debe estar correctamente funcionando y el usuario autenticado en el cliente
Secuencia de acciones:

1. El cliente se descarga la web de los ficheros estáticos del servidor.
2. El usuario se autentica correctamente y es presentado con la pantalla de selección de juegos
3. El usuario realiza la acción de crear una partida.

4. El servidor registra la creación de partida.

5. El cliente web solicita al servidor los datos de las publicaciones para esa partida y los jugadores.

Postcondiciones: Ninguna.

CU5: Selección de publicación

Una vez que el usuario ha escogido una partida el cliente se encarga de solicitar los datos de esa partida, incluyendo publicaciones y datos de los jugadores, en ese momento el usuario puede seleccionar una de dichas publicaciones.
Actores Implicados: Usuario, Sistema.
Precondiciones: El usuario debe haber seleccionado una partida.
Secuencia de acciones:

1. El cliente se descarga la web de los ficheros estáticos del servidor.

2. El usuario escoge un juego de todos los posibles.

3. El usuario es presentado con las publicaciones y escoge una de ellas.

4. El cliente web solicita al servidor los datos de dicha publicación, incluyendo los votos, comentarios y canción.

Postcondiciones: Ninguna.

CU6: Votación y/o comentario en una publicación

El usuario realiza una votación y/o un comentario para un publicación determinada.
Actores Implicados: Usuario, Sistema.
Precondiciones: El usuario debe haber seleccionado una partida y una publicación.
Secuencia de acciones:

1. El cliente se descarga la web de los ficheros estáticos del servidor.

2. El usuario se autentica correctamente, escoge un juego y selecciona una publicación.

3. El usuario publica un voto y/o un comentario a través del formulario.

4. El cliente web actualiza el estado de los votos y comentarios de la aplicación.

Postcondiciones: El voto debe quedar reflejado en la nueva lista de votos una vez que se ha actualizado la web.

CU7: Búsqueda de artista y canción

El usuario sigue el proceso de selección de artista y canción establecido por la aplicacón y es presentado con una selección de canciones a partir de búsquedas en las APIs de freebase y lastfm, del las cuales deberá elegir una.
Actores Implicados: Usuario, Sistema, Youtube, lastfm, freebase.
Precondiciones: El sistema debe estar correctamente funcionando y el usuario autenticado en el cliente
Secuencia de acciones:

1. El cliente se descarga la web de los ficheros estáticos del servidor.

2. El cliente se autentica correctamente y tras haber escogido un juego navega hasta la pantalla de administración de la cola de canciones.

3. El usuario introduce los datos de un artista en la búsqueda.

4. El servidor realiza la petición de búsqueda contra freebase o lastfm dependiendo de la prioridad establecida.

5. El cliente recibe la información y se la presenta al usuario, éste selecciona uno de los artistas e introduce la información de la canción a buscar.

6. El servidor recibe la nueva información y realiza la búsqueda de nuevo contra freebase o lastfm con los datos de artista y canción. En el caso de que existan coincidencias realizará una nueva búsqueda contra Youtube para finalmente devolver los videos disponibles para dicha canción al cliente.

7. El cliente recibe la lista de canciones del servidor y escoge una de ellas.

8. El servidor recibe la canción seleccionada y la guarda en la cola del usuario

9. El cliente recibe la respuesta del servidor y actualiza la página con la nueva cola de canciones.

Postcondiciones: La cola de canciones del usuario debe estar correctamente actualizada.

CU8: Consulta de datos de perfil del usuario

Una vez que el usuario se ha autenticado será capaz de consultar los datos de su perfil asćomo los de cualquier jugador del juego en curso.
Actores Implicados: Usuario, Sistema.
Precondiciones: El usuario debe haber seleccionado una partida.
Secuencia de acciones:

1. El cliente se descarga la web de los ficheros estáticos del servidor.
2. El cliente escoge un juego de todos los posibles.
3. El cliente navega hacia la pantalla de perfil de alguno de los jugadores incluido él mismo.
4. El cliente web solicita los datos de perfil del jugador escogido y los muestra al usuario.

Postcondiciones: Ninguna.

CU9: Administración de la API

Los usuarios designados como administradores tendrán acceso al área de administración de la aplicación. Una vez dentro podrá realizar operaciones de tipo CRUD (*Create, Read, Update, Delete*) sobre los recursos.
Actores Implicados: Usuario, Sistema.
Precondiciones: El usuario debe ser administrador.
Secuencia de acciones:

1. El cliente accede al área de administración de la aplicación.
2. El cliente web muestra al usuario todos los recursos susceptibles de ser modificados.
3. El cliente modifica los recursos a voluntad.

Postcondiciones: Los cambios realizados se reflejan sobre los datos de la base de datos.

CU10: El juego avanza diariamente

El script gameprocess.rb se ejecuta diariamente para que aparezcan nuevas publicaciones en los juegos en curso
Actores Implicados: Sistema.
Precondiciones: Ninguna.
Secuencia de acciones:

1. El sistema es alertado por una alarma de que debe lanzar el script de progreso.

2. El script se ejecuta y se producen los cambios en el sistema.

Postcondiciones: Los cambios deben verse reflejados en las partidas en curso.

3.4. Matriz de trazabilidad

A continuación se muestra una tabla en la que se muestran las relaciones existentes entre los casos de uso del sistema y los requisitos funcionales existentes, de tal manera que se pueda comprobar rápidamente que los casos de uso cubren la totalidad de los requisitos funcionales existentes.

Caso de uso	**RF1**	**RF2**	**RF3**	**RF4**	**RF5**	**RF6**	**RF7**	**RF8**	**RF9**	**RF10**	**RF11**	**RF12**	**RF13**	**RF14**	**RF15**	**RF16**	**RF17**	**RF18**	**RF19**
CU1	√																		
CU2		√																	
CU3			√	√		√													
CU4					√														
CU5							√	√	√										
CU6														√					
CU7										√	√	√	√		√	√			
CU8																		√	
CU9																			√
CU10																	√		

3.5. Arquitectura del sistema

Los actores que intervienen en la arquitectura son a nivel lógico los mismos que intervendrían en una aplicación web tradicional, salvando los nuevos tipos de clientes (móviles y tablets).

3.5.1. Servidor de autenticación

Es el responsable de realizar las labores de autenticación y autorización implementado alguno de los mecanismos de seguridad disponibles para ello. En mi caso he utilizado tecnología OAuth2 ya que es uno de los estándares más seguros de la web. Es importante separar a nivel lógico los servidores de aplicaciones de los de autenticación, puesto que de esta forma permitimos la escalabilidad horizontal de ambos por separado. Además el servidor de autenticación puede comunicarse con los diferentes actores del sistema independientemente de las conexiones que tenga el servidor REST o de aplicaciones. El servidor de autenticación se apoya en una base de datos mongoDB, cuyo diseño será explicado posteriormente, para almacenar los tokens de sesión de OAuth2, los usuarios registrados ó los niveles de autorización que tengan estos usuarios (administradores).

El protocolo OAuth2 está pensado para garantizar autenticación a través del protocolo HTTP mediante el intercambio de varios mensajes. Los clientes que deseen autenticarse deben tener primero **un par clientID, secret** con los que poder autenticarse. Es importante que este par **no debe ser público**, por lo que por ejemplo, una aplicación para el navegador no debería exponer dicho par. Los pasos que debe seguir un cliente para poder autenticarse son los siguientes:

- Primero el cliente debe realizar una petición POST a **/auth/token** con los siguientes parámetros POST:
  - username
  - password
  - HTTP Header tipo Authorization con el contenido: *Basic base64Encode(client-id:client-secret)*
- Una vez autenticado el servidor responderá con un token de autorización que el cliente deberá guardar para enviarlo en peticiones posterio-

res al servidor de aplicaciones mediante la siguiente header: *Authorization: Bearer 'token'*

Si se quiere dar de alta un nuevo cliente basta con generar un nuevo par de ID y secret para dicho cliente y guardarlo en la base de datos, para que a la hora de consultar, el servidor de autenticación sepa que ese cliente puede autenticarse.

3.5.2. Api REST

El API REST es una aplicación escrita en node.js que se mantiene activa **escuchando en el 1337**. Se encarga de procesar las peticiones de los diversos clientes y realizar las operaciones disponibles sobre los recursos, como leer, crear, modificar o borrar, además de otras operaciones especiales. La API se encarga también de para cada petición, consultar con el servidor de autenticación y realizar la operación solicitada o contestar con un error 500 (Not Autorized). Es importante comprender que una API REST no es más que un servido web tradicional orientado al manejo de recursos y el paso de variables a través e parámetros de la URL. En este sentido las APIs entienden el **protocolo HTTP** y pueden hablar el idioma que se especifique, que en mi caso ha sido JSON. La estructura de node.js esta comprendida por capas ó módulos (middleware) que van agregando funcionalidad a la plataforma:

- En primer lugar tenemos la capa principal de trabajo que ofrece node.js, la cual consiste en una serie de módulos que nos permiten crear conexiones **TCP/UDP**, interactuar con la entrada y salida de disco...etc, en definitiva se comporta como una librería estándar de un lenguaje de programación normal.
- A continuación se coloca la capa de connect.js que contiene funciones para el parseo de JSON, de cookies, de sesión entre otras.
- Por último tenemos express.js que añade las funcionalidades web necesarias para definir los puntos de acceso de una API REST de una forma sencilla abstrayéndonos de la creación manual de sockets y el parseo de los mensajes HTTP.

Este stack de node.js proporciona los recursos necesarios para poder desarrollar APIs rápidamente. La API REST contempla dos puntos de entrada principales para las aplicaciones cliente de administración en */admin/API* o la aplicación pública en *//API*.

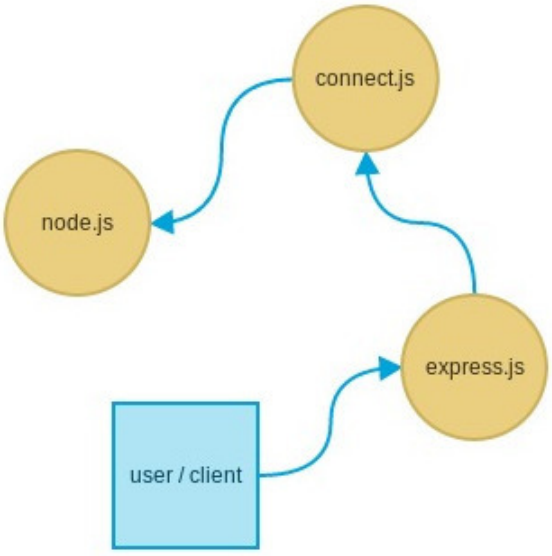

Figura 3.2: Stack de middleware node.js usado en la aplicación

3.5.3. Base de datos MongoDB

La base de datos es una de las piezas más importantes dentro de una aplicación. Es por ello que se debe prestar mucha atención tanto a la elección del sistema de gestión de base de datos como al modelado de los datos. En mi caso el software escogido ha sido MongoDB. MongoDB es un sistema de gestión de base de datos **NoSql orientado a documentos** con drivers disponibles para la mayoría de lenguajes de programación y plataformas, incluido node.js. A pesar de que no existen técnicas de reunión de tablas como en los motores relacionales y no es posible llevar a cabo una correcta normalización de los datos, las ventajas son lo suficiente superiores como para suplir a estas dos limitaciones. Los documentos se organizan en colecciones y estas a su vez en bases de datos. El formato de los datos es **BSON (Binary JSON)**, que a efectos prácticos se comporta de forma exactamente igual que JSON, ya que los drivers se encargan de transformar dicho BSON en texto plano ó JSON. Es posible organizar el modelo de datos de modo que algunos documentos hagan referencia a otros con el propósito de simular reuniones, aunque no es igual de eficiente que un join relacional. La principal ventaja de MongoDB es la sencillez de uso y el hecho de que al no estar normalizados los datos es fácil modificar el esquema cuando se produzcan cambios en el modelo de datos, lo cual es muy probable que ocurra.

MongoDB carece de un lenguaje de consultas estructurado igual que MySql u otras. Para realizar las consultas se utiliza un pseudo lenguaje basado en **javascript** que permite realizar las tareas más básicas. Son los drivers los que se encargan de abstraer dicha funcionalidad para ofrecer una capa de funciones de mayor nivel. En el caso de node.js el driver escogido ha sido mongoose.

MongoDB soporta dos características fundamentales para cualquier sistema de gestión de base de datos. La replicación y el sharding:

- **Replicación**: es una técnica que consiste en formar un conjunto de bases de datos para que actúen como si fueran una. En el caso de MongoDB es necesario al menos disponer de tres instancias que se hablen entre ellas y puedan negociar a quien corresponde el cargo de Maestro y a quienes los esclavos.

- **Sharding**: es el proceso de almacenar los documentos a través de múltiples instancias de mongo en distintas máquinas con el propósito de satisfacer las necesidades de escalabilidad horizontal debido al incremento

masivo de datos.

3.5.4. Nginx

Nginx es un servidor HTTP ó proxy inverso open source y de alto rendimiento. A diferencia de los servidores tradicionales como Apache, que utilizan threads o procesos para manejar las peticiones, Nginx utiliza el modelo de arquitectura asíncrona dirigida a eventos muy similar al que utiliza node.js que permite manejar múltiples más conexiones que los servidores orientados a threads. En mi caso he utilizado nginx como un servidor proxy inverso puesto que lo que nuestra aplicación necesita es exponer múltiples aplicaciones a través de un mismo puerto (el 80). A través de Nginx es posible configurar que urls deben acceder a distintas aplicaciones, incluso proporcionando mecanismos de clustering. La configuración de Nginx será vista con más detalle en capítulos posteriores.

3.5.5. Clientes Web

Los clientes Web se ejecutan desde el navegador por lo que nos enfrentamos a una restricción fundamental de seguridad: los navegadores no pueden hablar con distintos puertos o ips en una misma página. Evidentemente la solución usada es el empleo de un servidor proxy inverso que proporcione el mecanismo de comunicación como descrito anteriormente. Para nuestro sistema se han creado dos aplicaciones web cliente:

- Aplicación cliente principal: realiza consultas contra la parte de la API pública directamente y es la responsable de exponer la funcionalidad principal del juego al usuario.
- Aplicación de administración: reservada solo para administradores o usuarios con privilegios, realiza consultas contra la parte de administración de la API. Se encarga de manejar la funcionalidad más primitiva sobre los recursos directamente, como crear nuevas canciones o etiquetas...etc.

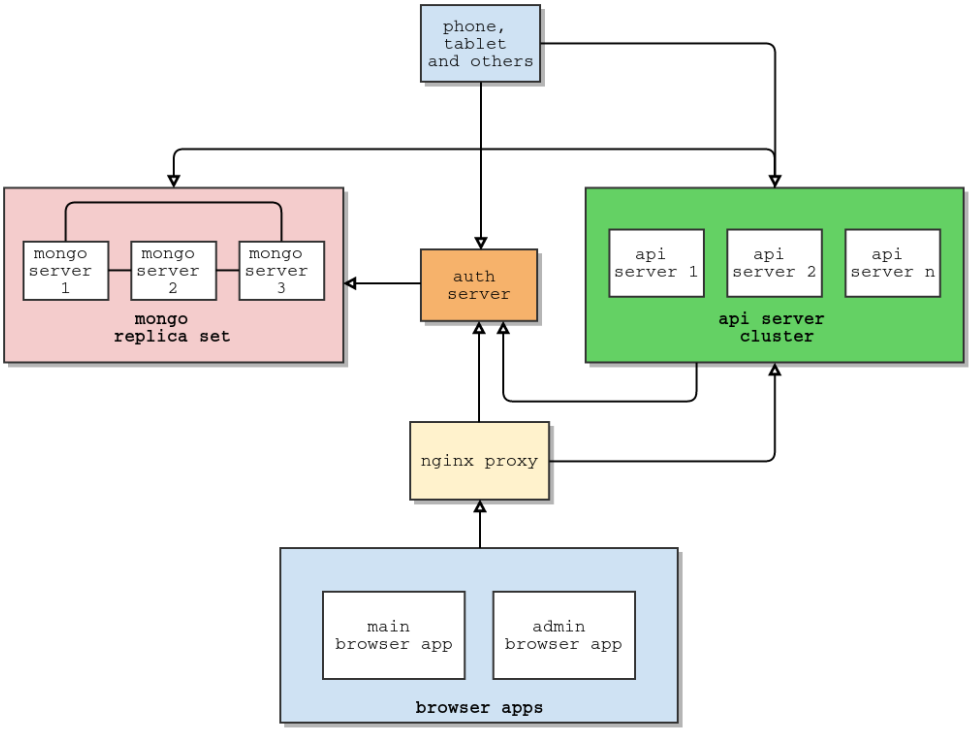

Figura 3.3: Visión general de la arquitectura del sistema

3.6. Especificación de la API

3.6.1. Api REST de la aplicación

En este punto se enumeran y se definen brevemente todos los puntos de entrada de la API REST pública que pueden ser accedidos por todos los clientes de la API. Los puntos de entrada de la parte de administración no se especifican puesto que son casos triviales. Todos los puntos parten de la url base **/API**.

- GET **/me**: devuelve los datos de perfil del jugador autenticado.
- GET **/players**: devuelve información acerca de los perfiles de los jugadores del juego en curso.
- GET **/players/:id**: igual que el anterior pero para un jugador concreto.
- GET **/game**: devuelve los datos del juego en curso.
- POST **/game**: permite crear un nuevo juego en el sistema y lo asocia a la lista de juegos del jugador que lo ha creado.
- GET **/rounds/(\w+)(/publications)**: devuelve la información de las publicaciones de una ronda determinada.
- GET **/publications/:id**: devuelve los datos de una publicación determinada.
- GET **/publications/:id/comments**: devuelve los comentarios de una publicación determinada.
- GET **/publications/:id/votes**: devuelve los votos de una publicación determinada.
- GET **/tags**: devuelve todas las etiquetas **activas** del sistema.
- GET **/queue**: devuelve la cola de canciones del usuario registrado en la aplicación.
- POST **/queue**: permite añadir una nueva canción a la cola de canciones del usuario.

- DELETE **/queue**: permite borrar una canción determinada de la cola de canciones del usuario.
- POST **/search/artists?artist=**: realiza una búsqueda en los servicios externos de lastfm y freebase por artista.
- POST **/search/songs?artistId=&song=**: realiza una búsqueda en los servicios externos de lastfm y freebase por artista y canción.
- GET **/songs?artist=&name=**: realiza una búsqueda de las canciones que cumplan los parámetros de entrada en el sistema.
- GET **/songs/youtube?artist=&name=**: realiza una consulta contra el API de youtube con los parámetros dados y devuelve una lista de videos.
- PUT/POST **/votes**: crea un nuevo voto con los datos proporcionados a través del cuerpo de la petición HTTP.
- PUT/POST **/comments**: crea un nuevo comentario con los datos proporcionados a través del cuerpo de la petición HTTP.

3.6.2. Api REST de autenticación

La definición de la API del servidor de autenticación es mucho más simple que la del resto de la aplicación, todas los puntos de entrada parten de la url base **/oauth**

- ALL **/token**: recibe como parámetros de entrada los datos descritos previamente para cumplir las normas del protocolo OAuth (clientID, secreto...) e intenta la autenticación de dicho cliente. Si el resultado es OK devuelve un nuevo token de sesión además de otros datos.
- POST **/logout**: invalida todos los tokens de sesión para el usuario autenticado.
- GET **/test**: comprueba si el usuario está autenticado a través del token de sesión.

Capítulo 4

Implementación

En este cAPItulo se explica paso a paso el proceso de implementación que se ha llevado a cabo para el sistema de acuerdo a los requisitos funcionales y no funcionales establecidos. Se mostrarán extractos de código y ficheros para que el lector pueda comprender en detalle los procesos y funcionalidades implementadas. La parte cliente de la aplicación ha sido diseñada, desarrollada y probada por ***Carlos Gómez Muñoz*** *(Desarrollo de una aplicación web cliente... El trabajo que se describe a continuación constituye el desarrollo de la parte servidor de la aplicación.*

4.1. MongoDB

La instalación de MongoDB en el sistema es relativamente sencilla, a continuación se describe el proceso de instalación para una distribución Ubuntu.

4.1.1. Paquetes

Los paquetes de MongoDB se pueden obtener directamente desde el repositorio de MongoDB. Los paquetes disponibles son los siguientes:

- **mongodb-org**: meta-paquete que instalará todos los paquetes listados a continuación.
- **mongodb-org-server**: paquete que contiene el demonio mongodb así como sus scripts de configuración y ficheros de arranque asociados.
- **mongodb-org-mongos**: contiene los demonios mongos.

- **mongodb-org-shell**: paquete que contiene la shell de mongo.
- **mongodb-org-tools**: contiene un conjunto de utilidades relacionadas con mongo, como por ejemplo *mongoimport*, *mongodump*, *mongorestore*.

4.1.2. Scripts y ficheros

El paquete mongodb-org incluye varios scripts de control, el más importante de ellos es el fichero de arranque de mongo que se encuentra localizado en **/etc/init.d/mongod**.
El fichero de configuración global de MongoDB se encuentra localizado en la siguiente ruta: **/etc/mongod.conf.**
El fichero de logs de mongodb se encuentra en **/var/log/mongodb/mongod.log**.

4.1.3. Instalación de MongoDB

El primer paso consiste en importar la clave publica del gestor de paquetes de MongoDB para garantizar la autenticidad y consistencia del paquete que vamos a descargar. Lo podemos realizar mediante el siguiente comando:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
   --recv 7F0CEB10
```

A continuación creamos el fichero /etc/apt/sources.list.d/mongodb.list mediante el siguiente comando:

```
echo 'deb
   http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist
   10gen' | sudo tee /etc/apt/sources.list.d/mongodb.list
```

Ahora ya podemos recargar la base de datos de paquetes para poder descargarnos el paquete mongodb-org del repositorio de MongoDB mediante el siguiente comando:

```
sudo apt-get update ; sudo apt-get install mongodb-org
```

A partir de este momento ya podemos manejar el demonio de mongodb como cualquier otro mediante las siguientes instrucciones:

```
sudo service mongod start|stop|restart|status
```

Para acceder a la shell de mongo basta con escribir **mongo** en un terminal.

4.2. Servidor de autenticación

La implementación del servidor de autenticación se ha realizado en node.js utilizando el paquete **node-oauth2-server**. Este paquete proporciona toda la lógica de parseo del protocolo OAuth2 y permite sobrescribir los métodos necesarios para almacenar los tokens y datos de usuario en diferentes bases de datos, de modo que podemos utilizar nuestra propia base de datos en mongo para almacenar los datos de sesión. La implementación del servidor se ha llevado a cabo en el proyecto **planum-server-oauth**, que contiene todos ficheros y paquetes necesarios para ejecutarlo, incluidos los tests.

4.2.1. Instalación

Lo primero que debemos hacer es instalar las dependencias, para ello ejecutamos los siguientes comandos en un terminal de Linux.

```
sudo apt-get install nodejs;
ln -s /usr/bin/nodejs /usr/bin/node;
sudo apt-get install npm;
npm install mocha -g
```

4.2.2. Funcionamiento

El proceso normal de funcionamiento del servidor consiste en conectar con la base de datos y mantenerse a la escucha de peticiones HTTP en el puerto indicado en la configuración. Para cada petición el servidor procesa los datos del cliente y decide si la autenticación ha sido correcta o no. Si lo ha sido devuelve una respuesta en formato JSON con los tokens de sesión y la información necesaria para renovar el token. La implementación del modelo

de datos se realiza en el fichero model.js. En él se definen la estructura de las clases de dominio en las que el servidor almacena los datos necesarios. El módulo **node-oauth2-server** está orientado de modo que las funciones del modelo se sobrescriben al estilo 'hooks' por lo que se deja al usuario la implementación relacionada con el manejo de base de datos.

```
1 model.getAccessToken = function (bearerToken, callback) {
2   OAuthAccessTokensModel.findOne({ accessToken: bearerToken
      }).populate('user').exec(callback);
3 };
```

Listing 4.1: Función para recuperar el token de usuario

El logout se realiza simplemente borrando todas las referencias a los tokens de sesión y a los tokens de refresco de la siguiente forma:

```
1  model.logout = function(token, callback) {
2    OAuthAccessTokensModel.findOne({ accessToken: token },
       function(err, tokenModel) {
3      if(err) { return callback(err) }
4      if(tokenModel == null) { return callback("No token was
         found to logout"); }
5      var user = tokenModel.user;
6      OAuthAccessTokensModel.find({ user: user }).remove().exec
         (function(err) {
7        OAuthRefreshTokensModel.find({ user: user }).remove().
           exec(function(err) {
8          callback(err);
9        });
10     });
11   });
12 }
```

Listing 4.2: Logout de usuarios

4.3. Api REST

El API REST es la pieza fundamental de la aplicación al fin y al cabo puesto que es la que satisface mayor número de requisitos funcionales. La implementación se ha realizado en node.js también, puesto que nos proporciona la sencillez de poder trabajar con los objetos en formato JSON sin necesidad de tener que realizar parseos de los datos que se intercambian con el resto

de aplicaciones del sistema. El API se apoya en la librería ó framework para node.js, express.js, que permite realizar la implementación de las llamadas de forma más cómoda, ya que nos proporciona las herramientas de parseo de rutas y variables. El sistema se encuentra divido en varias piezas principales:

- **domain**: compuesta por dos ficheros (*storage.js y domain.js*) se encarga de realizar las tareas relacionadas con el modelo de datos. Desde la definición formal de las clases de dominio mediante mongoose.js hasta las funciones más básicas de consulta, modificación y borrado de datos.

- **routes**: contiene los ficheros de rutas de la API tanto para la parte pública como para la administración. En estos ficheros se realiza la implementación de todas las entradas definidas para la API y los métodos auxiliares necesarios. Se comunica con el resto de paquetes, como el de dominio u otros, para manejar los datos, realizar consultas a terceras partes etc...

- **test**: en este paquete se ha realiza la implementación de los tests unitarios para las distintas partes de la aplicación. Se encuentra dividida en varios ficheros: *APITest.js, dataSourceTest.js, findTest.js, saveTest.js, utilTest.js, youtubeTest.js..*

- **music**: este paquete contiene las implementaciones para las consultas de los proveedores de datos externos (freebase y lastfm). Las implementación esta realizada de modo que los proveedores implementan una interfaz de funciones implícita, por lo que los paquetes que quieran usarlo se abstraen de las implementaciones concretas de cada proveedor.

- **youtube**: en este paquete se implementan las funcionalidades necesarias para la comunicación con el API de youtube y la búsqueda de vídeos con unos determinados parámetros.

- **scripts**: en este paquete se encuentran los ficheros de scripts de la aplicación. *Mockdata.js* se encarga de realizar un volcado de datos de prueba necesarios para pasar los tests, mientras que *gameprocess.rb* realiza la tarea de actualizar el juego.

- **node_modules**: contiene los módulos o librerías externas que se emplean en la aplicación. La lista de dichas librerías se encuentra packa-

ge.json, el cual define la librería y la versión que se instalará dentro de node_modules.

- **bin**: paquete muy simple que permite la instanciación de la aplicación mediante el comando *npm start*.

4.3.1. app.js

Este fichero es el **fichero principal** de la aplicación. Se encarga de resolver todas las dependencias de la aplicación en el orden en el que deben ser resueltas. Además tiene como responsabilidad la de servir como **proxy al servidor de autenticación** de modo que la clave del cliente y el secreto no queden expuestos al público. Cada vez que se realiza una petición a la API este fichero se encarga de garantizar que el cliente está correctamente autenticado y que además tiene autorización para acceder al área de la aplicación que esta solicitando. El fichero exporta su propia *app* de modo que el paquete bin pueda arrancarla mediante npm. La secuencia que sigue una petición al servidor cumple las siguientes normas:

1. La 'request' es procesada por todo el **middleware de connect.js** y express.js, es decir, se realiza un log de la request, se procesan las cabeceras y el cuerpo HTTP, se extraen las cookies de las cabeceras...etc.

```
1     app.use(favicon());
2     app.use(logger('dev'));
3     app.use(bodyParser.json());
4     app.use(bodyParser.urlencoded());
5     app.use(cookieParser());
```

Listing 4.3: Stack de middleware de connect.js

2. A continuación se comprueba si la petición debe ser redirigida hacia el servidor de autenticación, solo en el caso de que la request sea un POST hacia */oauth/token*.

3. En caso de que la petición no sea redirigida se comprueba la autenticación del cliente realizando una solicitud contra el servidor de autenticación en /oauth/test. Si el cliente está autenticado y tiene autorización para continuar se le asocia a la request el id de usuario por motivos de conveniencia mediante la sentencia **req.user = user**. En caso de que

el cliente no haya sido autenticado previamente el servidor responde con un código HTTP 401 Not Authorized.

```
1    request({ uri: oauthUrl + '/oauth/test', headers: {
        authorization: tokenHeader }}, function(err, msg,
        body) {
2      if(err || body === undefined) { return res.send
          (500, { error: "Internal server error" }); }
3      var bodyJson = JSON.parse(body);
4      var responseCode = bodyJson.ok;
5      var user = bodyJson.user;
6      var admin = bodyJson.admin || false;
7      if(responseCode == undefined || responseCode ==
          false || user == undefined) { return res.send
          (401, { error: "Not authorized" }); }
8      if(responseCode == true) {
9        req.user = user;
10       req.admin = admin;
11     } else {
12       return res.send(401, { error: "Not authorized" })
            ;
13     }
```

Listing 4.4: Autorización de la petición HTTP

4. Si el usuario estaba autenticado entonces se comprueba si la ruta corresponde con */API*, en cuyo caso se procede a delegar la request al fichero **API.js** para encuentre el método correspondiente.

5. Si la request no coincida con el patrón */API* se comprueba si tiene autorización para acceder al área de administración. Si la tiene entonces se comprueba si la request coincide con el patrón */admin/API*. En el caso de que coincida se delega de nuevo al fichero **admin.js**.

6. En caso de que no se hayan cumplida ninguna de las condiciones la petición llega al final del recorrido y es interceptada por el último método, el cual entrega un error *404 Not Found*.

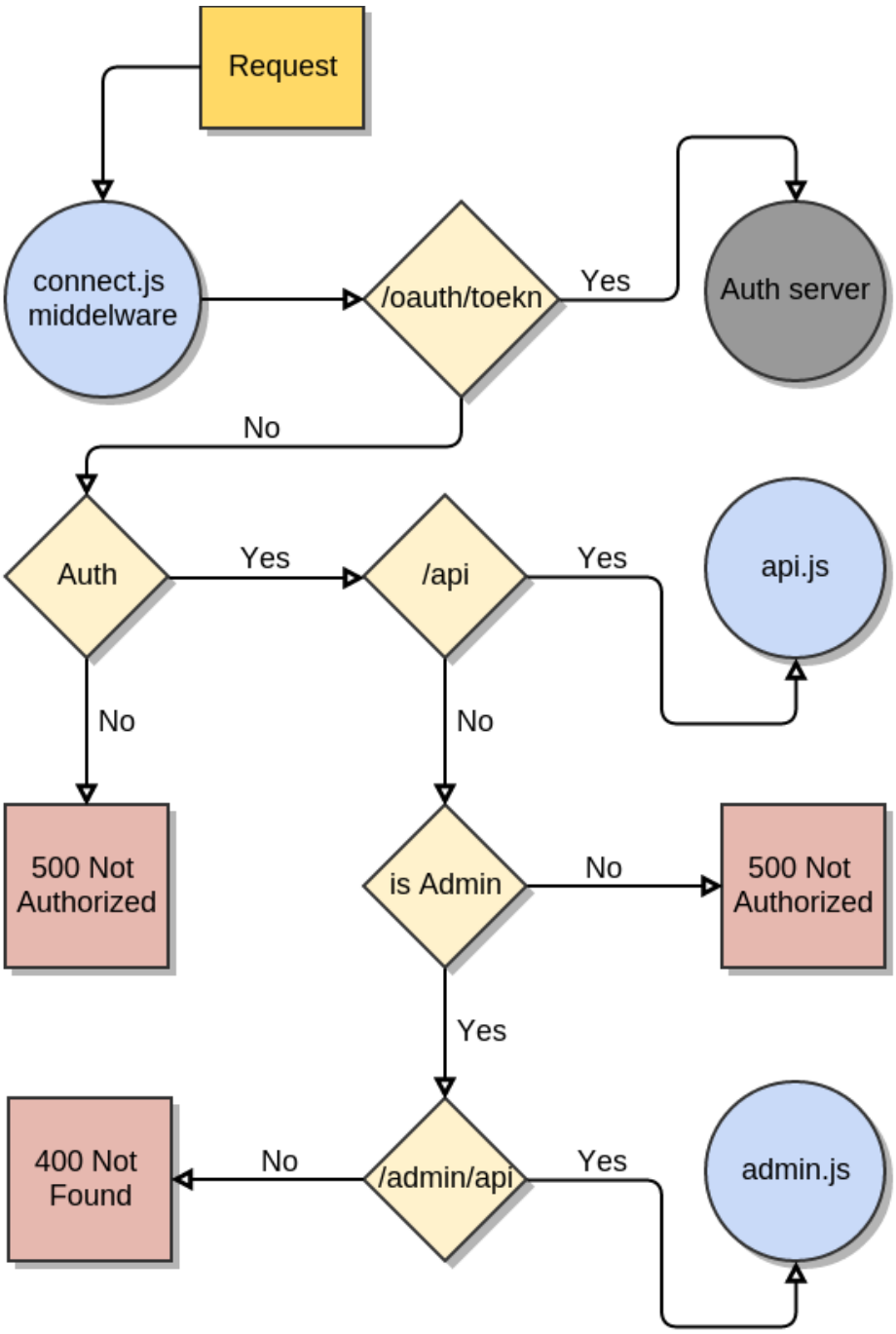

Figura 4.1: Secuencia de una request en app.js

4.3.2. config.js

Este fichero se encarga de mantener la configuración global de urls y parámetros del sistema de modo que si es necesario realizar cambios solo es necesario tocar este fichero. Entre otros se encuentra parámetros como la url del servidor de autenticación, el id de cliente de la API de youtube, la url de búsqueda de freebase...etc. Además en este fichero se definen la url y el puerto de la API en sí misma.

HATEOAS

Los HATEOAS (*Hypermedia as the Engine of Application State*) son una restricción de la arquitectura de aplicación REST cuyo principio consiste en que un cliente interactúa enteramente con una aplicación a través de las Urls que ésta le va proporcionando. Así pues cada vez que un cliente solicita un recurso, éste a su vez contiene los enlaces de hypermedia generados dinámicamente necesaria para poder continuar la navegación sin necesidad de conocer las URLS. Por ejemplo, un jugador puede estar asociado a un juego determinado, si el cliente pide los dato de un jugador se enviará también la relación del juego mediante HATEOAS de la siguiente forma:

```
1 {
2   jugador: {
3     nombre: 'Jorge',
4     juego: 'http://planum.com:80/API/juego/2sd34Fd234FFF
5   }
6 }
```

Listing 4.5: Ejemplo de json y HATEOAS

Para poder implementar el mecanismo de HATEOAS en los recursos es necesario modificar todas las respuestas de recursos que el servidor envíe a los clientes. Para ello he desarrollado una función dentro de *config.js*. Está función recibe como parámetro de entrada un objeto JSON, una url y una propiedad. Se reemplazaran todas las ocurrencias de dicha propiedad por la url dada más el id del recurso.

```
config.applyUrl = function(url, object, property) {
  if(object instanceof Array) {
    var counter = 0;
    object.forEach(function(item) {
      object[counter][property] = config.url + url.replace(':id', item[property]);
      counter++;
    });
  }else {
    object[property] = config.url + url.replace(':id', object[property]);
  }
}
```

Listing 4.6: Función de utilidad para HATEOAS

De esta forma las llamadas de la API que necesiten una transformación de HATEOAS simplemente tendrán que llamara a dicha función de la siguiente forma:

```
config.applyUrl('/publication/:id', votes, 'publication');
```

Listing 4.7: Ejemplo de traducción de HATEOAS

4.3.3. Domain

La carpeta domain contiene la funcionalidad relacionada con el almacenamiento de los datos y las conexiones con la base de datos MongoDB. Esta compuesto de dos ficheros:

model.js

Este fichero se encarga de establecer la conexión con MongoDB y definir las clases de dominio del sistema mediante el driver de MongoDB para node.js, mongoose. Mongoose se define como un ODM (*Object Document Mapper*. Se trata de un concepto muy similar al de un ORM, simplemente que orientado a documentos puesto que MongoDB trabaja con documentos. Mongoose permite definir un esquema para cada clase de dominio, en el cual establecemos los atributos que dicha clase debe tener. Además con mongoose es posible establecer relaciones entre las clases a través del parámetro *ref* de modo que MongoDB sea capaz de popular documentos a varios niveles.

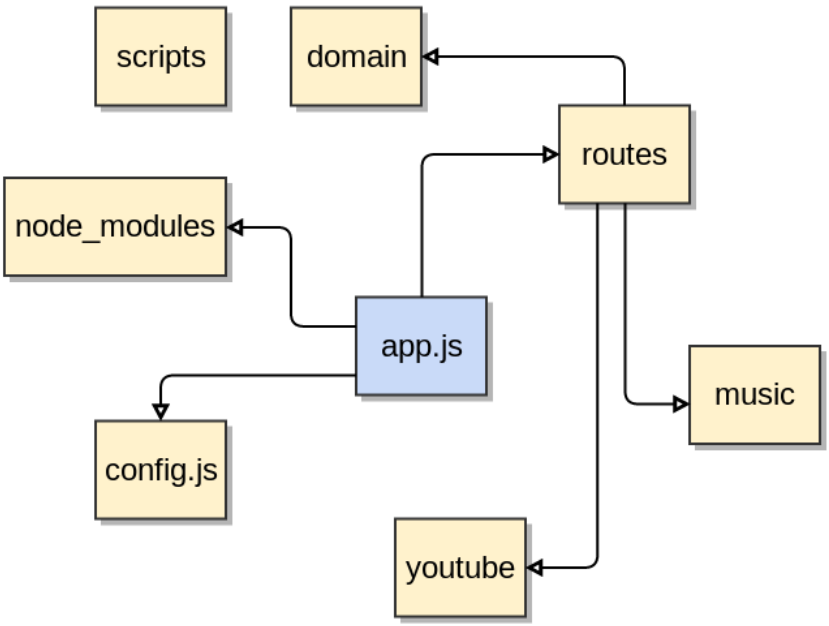

Figura 4.2: Visión general de los módulos de la aplicación

```
var Game = new Schema({
  name: { type: String },
  active: { type: Boolean },
  players: [{ type: ObjectId, ref: 'player' }],
  rounds: [{ type: ObjectId, ref: 'round' }],
  activeRound: { type: ObjectId, ref: 'round' }
});
```

Listing 4.8: Ejemplo de definición de esquema con Mongoose

storage.js

Las consultas en MongoDB se realizan a través del API de mongoose, y a diferencia de SQL no se trata de un lenguaje estructurado sino que el API expone una serie de métodos que pueden usarse en cadena para poder obtener los datos necesario. Por ejemplo, mientras en una consulta estilo SQL usaríamos *SELECT * FROM TABLE*, con mongoose debemos utilizar el método del drive *findAll*. Es posible popular lo documentos a varios niveles siempre y cuando se hayan definido referencias a dichos esquemas dentro de la definición del documento. En el caso del jugador es conveniente obtener los datos del juego asociado en vez de obtener el id del juego para posteriormente realizar la consulta con dicho id.

```
storage.findPlayer = function(id, populate, cb) {
  var query = model.Player.findById(id);
  if(populate) {
    query = query.populate('games');
  }
  query.exec(function(err, player) {
    if(err) return cb(err, null);
    return cb(false, player);
  });
}
```

Listing 4.9: Ejemplo de 'population' de documentos

Puesto que MongoDB no es capaz de realizar consultas estilo JOIN o similares, muchas veces nos vemos obligados a realizar consultas jerárquicas. Por ejemplo, cuando se piden las publicaciones de una ronda determinada, primero se debe buscar la ronda por id populando el campo de las publicaciones. Una vez populado se debe realizar una segunda consulta con el objetivo de popular de nuevo las canciones dentro de las publicaciones. En muchos

casos también es común la denormalización de los datos, puesto que una de las ventajas de NoSQL es que la no normalización de los datos no implica peor rendimiento o mayor dificultad en las consultas. En muchos casos es conveniente por ejemplo, almacenar una relación de objetos como un array de estos, en vez de crear una tabla de relación como debería hacerse en el caso de un motor de base de datos relacional.

```
1  storage.findPublicationsByRound = function(roundId, populate,
       cb) {
2    model.Round.findById(roundId, function(err, round) {
3      if(err) { return cb(err, null); }
4      if(!populate) {  return cb(false, round.publications); }
5      model.Publication.find({ '_id': { \$in: round.
           publications } }).populate('song').exec(function(err,
           publications){
6        if(err) { return cb(err, null); }
7        return cb(false, publications)
8      });
9    });
10 }
```

Listing 4.10: Ejemplo de consultas jerárquicas en mongoose

4.3.4. Music

El paquete music contiene todas las implementaciones de las consultas contra todas las posibles fuentes de datos relacionadas con la música, por ejemplo nombre de artista, canción, discos... El paquete está estructurado adaptándose al patrón de javascript **require**, de modo que existe un paquete **index.js**, cual es importado por el resto de paquetes que deseen utilizar la funcionalidad del paquete Music. Éste a su vez se comunica con el resto de paquetes para poder hablar con las APIs de datos musicales. El paquete expone dos métodos:

- **searchSong**: realiza una consulta de canciones sobre las bases de datos disponibles. Recibe como parámetro el nombre de la canción a buscar así como el id del artista al que pertenece.
- **searchArtist**: realiza una consulta de artistas. Recibe como parámetro de entrada el nombre del artista.

```
1 var MusicInfo = require('../music');
2 var musicInfo = new MusicInfo();
```

Listing 4.11: Ejemplo del uso del patrón 'require'

freebase.js

Freebase es una base de datos al estílo ontologá de personas, lugares y 'cosas' en general incluyendo artistas y canciones. Las consultas contra freebase se realizan mediante un lenguaje de consultas llamado MQL (*Metaweb Query Language*), similar al lenguaje de ontologías SPARQL, usando comunmente en RDF. Utiliza objetos JSON como consultas a través del protocolo HTTP utilizando la URL base *https://www.googleAPIs.com/freebase/v1/mqlread?query=*.

```
1   var query = [{
2     "type": "/music/artist",
3     "name~=": artistQuery,
4     "ns0:name": null,
5     "id": null,
6     "genre": [],
7     "label": [],
8     "/common/topic/image": [{}],
9     "sort": "ns0:name",
10    "limit": 5
11  }];
```

Listing 4.12: Consulta en MQL

lastfm.js

A diferencia de freebase el API de lastfm es una API **REST tradicional** que utiliza varios métodos de entrada a partir de las rutas para obtener los datos necesarios. Es necesario registrarse en lastfm y obtener las acreditaciones de id de cliente y secreto de la API. Para la comunicación con dicho he API he usado un paquete de node.js implementado por terceros llamado **lastfm-node**, que permite abstraer la lógica de requests y autenticación y realizar las llamadas directamente.

```
1 var LastFmNode = require('lastfm').LastFmNode;
```

Listing 4.13: Instancia del paquete de comunicación con node.js

4.3.5. Youtube

El paquete de Youtube contiene la lógica necesaria para realizar las comunicaciones con el API de vídeos de youtube. El paquete también esta organizado entorno al patrón 'require' por lo que la lógica también se expone a través del fichero *index.js* dentro del paquete. La implementación es muy simple, se crea una URL a través del método *createUrl* del fichero *util.js*, el cual coloca la URL base del API de youtube y agrega los parámetros get necesarios para realizar la consulta. Las consultas contra el API de youtube se realizan indicando los parámetros del vídeo que se desean recoger, como por ejemplo id, thumbnail...etc, además de los parámetros de búsqueda necesarios.

```
1  Youtube.search.list({
2    'part': 'id, snippet',
3    'q': artist + ' ' + name
4  }, function(err, data) {
5    if(err) { return res.send(500, { error: 'Unexpected
       server error' }); }
6    res.json(parseYoutubeData(data));
7  });
```

Listing 4.14: Ejemplo de consulta mediante el paquete Youtube

4.3.6. Routes

El paquete routes contiene los ficheros que implementan la funcionalidad fundamental de la API, es decir como se debe responder ante una llamada determinada. El módule de node.js, express.js, es capaz de analizar la url de modo que tan solo es necesario especificar una función para cada entrada de la API. Para poder implementar dichas funciones es necesario importar el objeto *router* de express.js de la siguiente forma:

```
1 var router = express.Router();
```

Una vez importado ya es posible comenzar con la implementación de los métodos de la API. El modo de hacerlo es el siguiente:

```
1 router.get('/players/:id', function(req, res) {
```

Express.js es capaz de interceptar las solicitudes tipo GET y analizar la URL para comprobar a través de expresiones regulares si coincide con el patrón

especificado en la definición del método. En el caso del extracto superior, es posible acceder a la variable mediante **req.params.id**. También es posible acceder a las variables tipo GET mediante **req.query**. Los parámetros tipo POST son accesibles mediante **req.body**, que es en esencia acceder al cuerpo de la solicitud. También es posible introducir expresiones regulares como por ejemplo:

```
1 /^\/rounds\/(\w+)(\/publications)*\$
```

En este caso la expresión permite obtener los datos de las publicaciones de una ronda determinada sin necesidad de especificar */publications*. En definitiva este fichero contiene las implementaciones de todos los métodos definidos previamente de la API. Es importante tener en consideración que node.js se trata de un lenguaje asíncrono por lo que siempre debemos devolver algo en las funciones de la API, es decir, no basta con responder a la request, puesto que ese código se ejecutaría al mismo tiempo que el resto:

```
1 router.post('/game', function(req, res) {
2   var user = ObjectId(req.user);
3   var game = req.body;
4   storage.createGameForPlayer(user, game, function(err, games
       ) {
5     if(err) { return res.send(500, { error: 'Unexpected
         server error' }); }
6     res.json(games);
7   });
8 });
```

4.3.7. Scripts

El paquete de scripts contiene scripts en javascript y ruby necesarios para el testeo y funcionamiento de la aplicación.

mockdata.js & createdb.js

Son scripts en 'javascript' escritos específicamente para el API de consultas de MongoDB. Para ejecutar ambos scripts se debe realizar la siguiente sentencia en un terminal:

```
mongo mockdata.js
```

Mientras que createdb.js se encarga de crear las colecciones en la base de datos, mockdata.js inserta los datos de testing para que los test unitarios puedan pasar.

gameprocess.rb

Es un script en ruby que modifica los datos del sistema y realiza las operaciones necesarias para que haya progreso en el juego, es decir que se actualicen las publicaciones y se creen nuevas rondas. Utiliza el driver de MongoDB para ruby para realizar las consultas y modificaciones:

```
1 require 'mongo'
2 include Mongo
3
4 mongo_client = MongoClient.new("mongodb", 27017)
```

A grandes rasgos en proceso que lleva a cabo el script es iterar sobre todos los juegos activos y crear una nueva publicación de para la última ronda disponible de cada juego. A continuación se extrae una nueva canción de las colas de jugadores que no hayan publicado una canción y se asocia con la nueva publicación.

```
1 # Create the new publication
2 new_publication = { 'song' => song_id, 'round' =>
     active_round['_id'], 'player' => player_id, 'date' => Time
     .now.getutc }
3 new_publication_id = publications_coll.insert(new_publication
     )
4
5 # Push the publication into the active round
6 rounds_coll.update({ :_id => active_round['_id'] }, { '\$push
     ' => { 'publications' => new_publication_id } })
```

4.3.8. Administración

La adminstración del sistema es una aplicación escrita en javascript apoyándose en el framework Ember.js para realizar operaciones estílo CRUD sobre los recursos del sistema. La aplicación se comunica con los servidores de la API REST excepto que accede a través de la URL /admin/API por lo que express.js intercepta la petición y la destina al fichero *admin.js*

Aplicación cliente

La aplicación cliente está realizada usando el framework Ember.js. Ember.js es un framework cliente estilo MVC. Permite crear aplicaciones escalables estilo single-page applications por lo que es perfecto para aplicaciones del estílo administrativo. Con ember.js se pueden definir los modelos de nuevo en el cliente y automatizar las llamadas Ajax. La apliación está dividida en cuatro ficheros principales:

- **App.js**: contiene la declaración de rutas del sistema.

```
1 this.resource('home', { path: '/' });
2 this.resource('players', function() {
3   this.resource('player', { path: '/:player_id' });
4   this.resource('player.new', { path: '/new' });
5 });
```

Listing 4.15: Extracto de la definición de rutas ember

- **controller.js**: definición de modelos y de controladores.

```
1   App.Game = DS.Model.extend({
2   name: DS.attr('string'),
3   active: DS.attr('boolean'),
4   players: DS.hasMany('player', { async: true })
5 });
```

Listing 4.16: Definición del model Game

- **routes.js**: implementación de las rutas definidas previamente.
- **auth.js**: contiene la funcionalidad necesaria para comunicarse con el servidor de autenticación y almacenar los datos de respuesta en el almacenamiento local del navegador.

```
1     authenticate: function(user, password, cb) {
2     var data = { grant_type: 'password', username: user,
          password: password };
3     var request = \$.ajax({
4       type: 'POST',
5       beforeSend: function(request) {
6         request.setRequestHeader('Content-Type', '
              application/x-www-form-urlencoded');
7       },
```

```
8         url: '/oauth/token',
9         data: data
10      });
```

Listing 4.17: autenticación a través del cliente

Aplicación servidor

La aplicación servidor es básicamente una aplicación REST que contiene las operaciones CRUD para todos los recursos que puedan ser administrados. El acceso público a esta aplicación implica numerosos riesgos de seguridad por lo que tan solo pueden acceder aquellos usuarios del sistema que sean administradores. La lógica del servidor está implementada dentro del paquete routes, en el fichero admin.js. A continuación se muestra un extracto de dicho código:

```
1 router.put('/players/:id', function(req, res) {
2   var data = req.body.player;
3   data._id = req.params.id;
4   savePlayer(data, function(err, player) {
5     if(err) { return res.send(500, { error: 'Unexpected
          server error' }); }
6     res.json({ player: player });
7   });
8 });
```

Listing 4.18: Extracto de código de admin.js

4.4. Implantación y Despliegue

A continuación se describe el proceso llevado a cabo para el despliegue e instalación necesario. Desde la instalación de las dependencias hasta la creación de los scripts de arranque de las APIs, pasando por la configuración de los proxies y la configuración de Amazon EC2 y las políticas de seguridad.

4.4.1. Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) es una parte central de la plataforma de computación en la nube de Amazon denominada Amazon Web Services (AWS). EC2 permite a los usuarios rentar computadores virtuales en

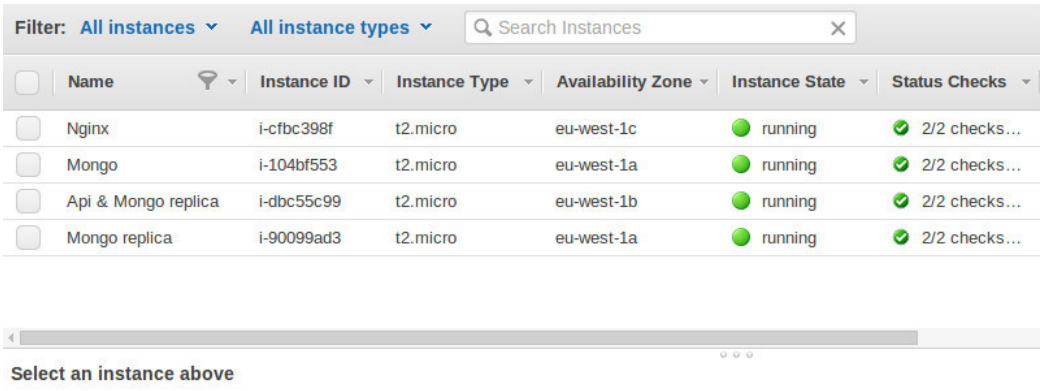

Figura 4.3: Captura de las instancias creadas en Amazon EC2

los cuales poder correr sus propias aplicaciones. Este tipo de servicio supone un cambio en el modelo informático al proporcionar capacidad informática con tamaño modificable en la nube, pagando por la capacidad utilizada. En lugar de comprar o alquilar un determinado procesador para utilizarlo varios meses o años, en EC2 se alquila la capacidad por horas. EC2 permite el despliegue escalable de aplicaciones proveyendo un servicio Web a través del cual un usuario puede montar una Imagen de Máquina Amazon para crear una máquina virtual, llamada por Amazon ïnstancia", la cual contendrá cualquier software deseado.

En mi caso he creado 4 instancias dentro del sistema de Amazon:

- **Nginx**: contiene la instancia del proxy inverso y las aplicaciones cliente como ficheros estáticos que nginx se encarga de servir. También contiene una instancia del servicio de la API REST planum.
- **Api & Mongo replica**: Actúa como respaldo para el 'replica set' de mongo además de otra segunda instancia del servicio de API REST planum.
- **Mongo**: Es el servidor maestro del 'replica set' de MongoDB.
- **Mongo replica**: Tercera réplica del 'replica set'.

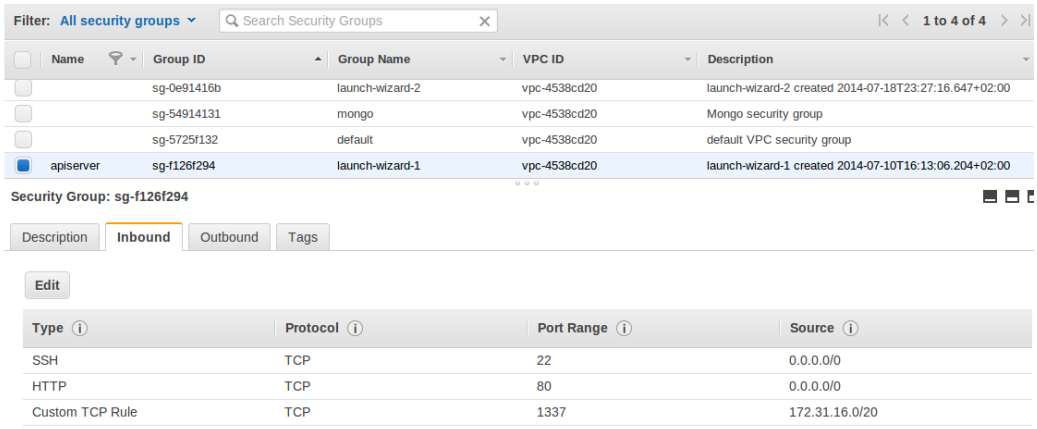

Figura 4.4: Captura de los Security Groups creados en Amazon EC2

Configuración de seguridad de Amazon EC2

Amazon EC2 permite definir una serie de Security Groups de modo que las distintas instancias puedan asociarse con los distintos Security Groups disponibles. Los Security Groups a su vez permiten definir una tupla de protocolo de comunicación y puerto para diferentes orígenes, por lo que podemos asegurar por ejemplo, que solo van a ser capaz de comunicarse con las instancias de MongoDB los servidores de API a través de la red interna y no cualquier otro host de internet.

4.4.2. Nginx

Nginx es un servidor proxy inverso de alto rendimiento que permite resolver los problemas modernos de escalabilidad de hoy en día. En nuestro caso Nginx ha sido usado tanto como servidor proxy inverso como servidor HTTP.

Proxy inverso

Un servidor proxy inverso en un servidor proxy que recupera recursos desde el cliente de uno o más servidores que se encuentran por detrás del proxy. Mientras que un servidor proxy normal actúa como intermediario entre los clientes para que puedan acceder a los recursos de internet, un servidor

proxy actúa como intermediario para únicamente sus servidores asociados, de modo solo queda expuesto como una sola máquina al resto de la red. En nuestro caso ha sido necesario configurar un proxy inverso ya que hay dos aplicaciones funcionando en distintos puertos además del servidor de ficheros estáticos, por lo que no todos pueden escuchar el puerto 80. De esta forma el servidor proxy inverso (Nginx) captura las peticiones de los clientes y las redirige a las diferentes aplicaciones o servidores detrás de la red. Por ejemplo, cuando un cliente realiza una petición a planum.com/oauth/token, en realidad es nginx el que captura dicha petición, y a continuación la enruta hacia el servidor de autenticación.

La configuración de nginx para que actué como servidor proxy inverso se realiza mediante la directiva *proxy_pass*.

```
location /oauth/logout {
    proxy\_pass http://localhost:3000;
  }

  location /oauth/test/ {
    proxy\_pass http://localhost:3000;
  }

  location /oauth/ {
    proxy\_pass http://API;
    proxy\_set_header Host \$http\_host;
  }
```

Para proporcionar el balanceo de carga entre distintos servidores de la API nginx nos permite definir un objeto *upstream* en el que se declaran los servidores a los que debe balancear. Por defecto se realiza un balanceo mediante Round Robin, pero existen diferentes opciones para balancear por pesos por ejemplo.

```
location /admin/API/ {
    proxy\_pass http://API;
  }

  upstream API {
    server localhost:1337;
    server APIserver:1337;
  }
```

Se han definido tres diferentes 'sites' a través de los cuales se pueden acceder a las aplicaciones:

- **planum.com**: dominio principal, redirige hacia el resto de aplicaciones.
- **oauth.planum.com**: dominio de autorización, redirige hacia el servidor de autenticación.
- **API.planum.com**: dominio de APIs, redirige hacia los servidores de las APIs.

Servidor HTTP

Además del servidor de proxy inverso, también hemos empleado Nginx para servir los siguientes ficheros estáticos:

- Aplicación cliente de planum directamente en la raíz /.
- Aplicación de administración de planum en **/adm**.
- Imágenes en **/img**

Para ello se han definido varios objetos de configuración en la configuración de nginx mediante la directiva *location* y las opciones de **alias** y **root**.

```
location /css/ {
        root /var/www/planum/webapp/app;
    try\_files \$uri \$uri/;
}

location /js/ {
        root /var/www/planum/webapp/app;
    try\_files \$uri \$uri/;
}

location /partials/ {
        root /var/www/planum/webapp/app;
        try\_files \$uri \$uri/;
}

location /adm {
        alias /var/www/planum/admin-app;
        # try\_files \$uri \$uri/ index.html;
}

location / {
        root /var/www/planum/webapp/app;
        try\_files \$uri \$uri/ index.html;
}

location /img/ {
        try\_files \$uri \$uri/;
}
```

4.4.3. Planum como servicio

Las aplicaciones node.js pueden lanzarse directamente a través del terminal mediante la sentencia *node app.js*. Otra opción consiste en definir un servicio a través de la interfaz **upstart** de ubuntu para obtener todas las ventajas de un servicio frente a una aplicación standalone. Con un servicio nuestras aplicaciones se lanzaran automáticamente al arranque de la máquina, además podemos parar y arrancar las aplicaciones sin tener que recuperar

el id proceso utilizando la sintaxis de servicios de ubuntu *service planum start/stop/status/restart*.

Upstart es en esencia un reemplazo basado en eventos para el antiguo demonio init. Permite gestionar tareas y servicios cuando se producen diferentes señales en el sistema, como por ejemplo el arranque y el apagado. Para definir un servicio en upstart basta con escribir un pequeño script upstart con extensión *.conf* u colocarlo en */etc/init*.

En nuestro caso se han definido dos scripts:

- **planum.conf**: script de arranque de la API REST situada en */home/ubuntu/planum/API*.
- **planum-oauth.conf**: script de arranque del servidor de autenticación situado en */home/ubuntu/planum/oauth*.

Forever

Forever es una aplicación de terminal que permite que las aplicaciones de node.js se ejecuten continuamente. Puesto que las aplicaciones javascript son interpretadas, cuando se produce un fallo, el interprete termina su ejecución y la aplicación produce un error. En este sentido son menos tolerantes a fallos que las aplicaciones compiladas. Es por tanto necesario utilizar el script forever para garantizar que ante un error no controlado se volverá a levantar la aplicación automáticamente. Su utilización es muy simple, basta con arrancar las aplicaciones node.js con forever de la siguiente forma *forever app.js*.

A continuación se muestra el fichero de upstart **planum.conf**:

```
description "planum auth and web"
author      "Jorge Madrid"

env PATH=/bin:/usr/bin:/usr/local/bin
env RUNUSER=ubuntu
env HOME=/home/ubuntu

start on (local-filesystems and net-device-up IFACE=eth0)

stop on shutdown

expect fork

# Automatically Respawn:
respawn
respawn limit 99 5

script

    exec forever -a -l \$HOME/logs/API.log -o
   \$HOME/logs/API.log -e \$HOME/logs/API.err.log start -c "npm
   start" \$HOME/planum/API/
end script

pre-stop script
    exec forever stop -c "npm stop" \$HOME/planum/API
end script
```

4.4.4. Replicación en MongoDB

La replicación en los sistemas de bases de datos proporcionan redundancia y disponibilidad de los datos. Con la existencia de múltiples copias de los datos en distintos servidores, la replicación protege la base de datos de la caída de un solo servidor. La replicación también permite incrementar el rendimiento de las operaciones de lectura puesto que varios clientes pueden atacar a los distintos miembros de la replica para leer, no para escribir.

Un 'replica set' en mongo está formado por tres servidores, un maestro y dos esclavos que se comunican con un 'heartbeat' cada cierto intervalo

de tiempo configurable. El maestro actúa como punto de acceso para todas las operaciones de escritura y automáticamente replica dichas operaciones al resto de miembros de la replica para garantizar la consistencia.

El primer paso para proporcionar la replica es levantar dos instancias más en Amazon EC2 o aprovechar alguna de las ya existentes y realizar la instalación del servidor de mongo. Una vez realiza deben**habilitarse los puertos** en la configuración de seguridad de Amazon. Es muy importante también definir los nombres de las máquinas en los ficheros hosts.

```
172.31.43.79 mongodb
172.31.47.4 mongodb2
```

El siguiente paso es modificar el fichero ***/etc/mongodb.conf*** para que la instancia de mongo apunte a la replicaset planum. Un error común es no modificar el parámetro de configuración por defecto de mongo para que no sólo acepte conexiones internas, el cual es el comportamiento por defecto. Para definir los miembros de la replica es necesario conectarse con un cliente de mongo al servidor maestro del conjunto y realizar los siguientes pasos:

- **rs.initiate**: instancia la replica set.
- Agregar los hosts mediante **rs.add("mongodb2")**
- **rs.status**: Permite comprobar el estado del 'replica set'.

```
planum:PRIMARY> rs.status()
{
        "set" : "planum",
        "date" : ISODate("2014-08-25T08:36:09Z"),
        "myState" : 1,
        "members" : [
                {
                        "_id" : 0,
                        "name" : "mongodb:27017",
                        "health" : 1,
                        "state" : 1,
                        "stateStr" : "PRIMARY",
                        "uptime" : 1627307,
                        "optimeDate" :
    ISODate("2014-08-23T09:56:22Z"),
                        "electionTime" : Timestamp(1407330059,
    1),
                        "self" : true
                },
                {
                        "_id" : 1,
                        "name" : "mongodb2:27017",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 1620613,
                        "lastHeartbeat" :
    ISODate("2014-08-25T08:36:07Z"),
                        "pingMs" : 0,
                        "syncingTo" : "mongodb:27017"
                },
                {
                        "_id" : 2,
                        "name" : "APIserver:27017",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 1620526,
                        "lastHeartbeat" :
    ISODate("2014-08-25T08:36:08Z"),
                        "pingMs" : 1,
                        "syncingTo" : "mongodb:27017"
                }
        ]
}
```

Capítulo 5

Pruebas

En este capítulo se explican las características más significativas del plan de pruebas, incluyendo un breve repaso de los conceptos básicos de la realización de pruebas de software. A continuación se presenta una panorámica general del diseño de la batería de pruebas del sistema. En este punto se mencionan también los principales problemas encontrados a la hora de desarrollar pruebas sistemáticas para este sistema.

La metodología seguida en todo el proyecto ha sido **TDD**, es decir, primero se realizan las pruebas unitarias, las cuales definen el comportamiento esperado y a continuación se implementa la funcionalidad para que dichos tests puedan pasar.

Para las pruebas se ha utilizado el framework para node.js, mocha, el cual permite realizar pruebas unitarias automáticas incluso con funcionalidad asíncrona. El comportamiento de las pruebas está más orientado a comportamiento que a funcionalidad por lo que suele definirse como **BDD** (*Behavioural Driven Development*). También se ha empleado la librería *should.js* que permite realizar 'assertions' de manera expresiva y simple.

5.1. Api REST

Los tests de la api REST están divididos en varios ficheros:

- **apiTest.js**: Son los tests unitarios del código de la api implementado en el fichero api.js.
- **findTest.js**: Tests unitarios relacionados con la lectura de datos de

mongo.

- **saveTest.js**: Tests unitarios relacionados con la escritura de datos de mongo.
- **dataSourceTest.js**: Test unitarios para la implementación de la comunicación y extracción de datos de las apis
- **youtubeTest.js**: Tests unitarios para la comunicación con el api de vídeos de Youtube.
- **utilTest.js**: Fichero que contiene los tests unitarios para el fichero util.js.

```
1  it('should GET a game', function(done) {
2    bootstrapRequest('/api/game', 'get')
3    .end(function(err, res) {
4      if(err){ console.log(err); }
5      var body = res.body;
6      res.status.should.equal(200);
7      body.should.be.a('object');
8      assertProperty(body, 'string', 'name');
9      assertProperty(body, 'Boolean', 'active');
10     assertProperty(body, 'string', 'activeRound');
11     assertProperty(body, 'array', 'rounds');
12     assertProperty(body, 'array', 'players');
13     done();
14   });
15 });
```

Listing 5.1: Extracto de un test de apiTest.js

```
1  it('should query for song in freebase', function(done) {
2    var artist = { id: '/dataworld/freeq/job_a27f2df3-7839-4
       e40-bbb2-
       acfbb3c39624_var_en_wikipedia_org_wiki_Hozier_$0028musician$0029
       '};
3    freebaseSource.searchSong('Take Me To Church', artist,
       function(data) {
4      data.should.be.a('array');
5      data.length.should.be.above(0);
6      data[0].should.have.a.property('freebaseId');
7      done();
8    });
9  });
```

Listing 5.2: Extracto de dataSourceTest.js

5.2. Servidor de autenticación

Las pruebas para el servidor de autenticación se han desarrollado de forma muy similar a las de la api REST. Se ha seguido la misma metodología de desarrollo de pruebas TDD. En este caso solo existe un fichero de pruebas *test.js*, que contiene todos los tests de la lógica del servidor.

```
1  describe('POST /oauth-token', function() {
2    it('should responde with 400 and invalid_parameter because
       no Content-Type header was set', function(done) {
3      request(app)
4        .post('/oauth/token')
5        .set('Accept', 'application/json')
6        .end(function(err, res) {
7          if(err){console.log(err)};
8          res.status.should.equal(400);
9          var body = res.body;
10         body.should.be.a('object');
11         body.code.should.equal(400);
12         body.error.should.equal('invalid_request');
13         done();
14       });
15 });
```

Listing 5.3: Extracto de un test de test.js

5.3. Pruebas de integración

Pruebas integrales o pruebas de integración son aquellas que se realizan en el ámbito del desarrollo de software una vez que se han aprobado las pruebas unitarias.

Todas las pruebas de integración han sido realizadas con Carlos Gómez (desarrollador de la aplicación cliente de la api). Las pruebas han consistido en la implementación e integración progresiva de las funcionalidades de la api con las de la aplicación cliente.

- Pruebas de registro y login de **usuarios**: Se comprueba que los usuarios pueden registrarse y autenticarse en la aplicación cliente.
- Pruebas de usabilidad y rendimiento: Se comprueba que el usuario puede usar el sistema de forma óptima.
- Pruebas de selección de canciones: Se comprueba que el usuario es capaz de agregar canciones a la cola mediante el proceso de selección implementado en la aplicación cliente, la cual realiza las consultas contra la api REST.

Capítulo 6

Resultados y Conclusiones

Como se explico al comienzo de la memoria el objetivo principal del proyecto era crear un sistema escalable que responda a las necesidades de disponibilidad y que cumpla los requisitos funcionales de la aplicación. En nuestro caso concreto la aplicación responde ante los requisitos de un juego en el que el modelo de dominio es la música y los usuarios los jugadores que deseean descubrir nuevas canciones. Sin embargo el 'stack' o 'pila' de tecnologías empleadas puede ser fácilmente extensible para cualquier otro ámbito de dominio o aplicación.
Incluso es perfectamente posible tomar sólo aquellas piezas de la aplicación que se deseen a modo de ejemplo para ser analizadas y empleadas en cualquier otro proyecto.

6.1. Resultados

El resultado de emplear metodología ágil junto con desarrollo dirigido con pruebas a sido satisfactorio en nuestro caso, pero no siempre tiene por que adecuarse al proyecto que se vaya a desarrollar. La metodología ágil es muy útil siempre y cuando el grupo de personas sea reducido y los requisitos vayan a cambiar a lo largo del desarrollo, lo cual suele ser lo normal en los desarrollos de software hoy en día.

Por otra parte, el empleo de nuevas tecnologías es siempre una decisión arriesgada a la hora de resolver problemas, sin embargo los beneficios que puede proporcionar una vez se ha superado la curva de aprendizaje suelen ser muy numerosos, por ejemplo, el desarrollo en NodeJS puede ser complejo

y chocante, pero una vez que se comprenden los principios esenciales se tiende a agilizar mucho el proceso puesto que javascript es un lenguaje mucho más dinámico y menos estrícto que los que normalmente se escogen para el desarrollo de aplicaciones web. Del mismo modo, las bases de datos NoSql pueden ser consideradas peligrosas por algunos puristas pero las ventajas que proporcionan una vez se han roto los esquemas de normalización son tantas que deberían ser valoradas de forma más positiva a la hora de escoger un sistema de gestión de base de datos.

En cuanto al despliegue en Amazon y la administración de sistemas, es importante considerar que es una tecnología en pleno apogéo, y cada vez usada por más empresas. Es por ello, que un conocimiento básico de como funcionan los servicios Cloud de Amazon puede ser un factor clave a la hora de desplegar aplicaciones o servicios web.

6.2. Lineas futuras

A continuación se describen algunas de las lineas futuras a continuar con el proyecto.

6.2.1. VIP

La implantación de una tecnología de tipo VIP permite que los sistemas resolver el problema de alta disponibilidad frente a la caida del servidor proxy o servidor principal. En este escenario, el servidor proxy cuenta con una segunda instancia y un software específico que permite la comunicación entre ambas. Esto permite simular una ip Virtual que actua como receptor lógico de las peticiones HTTP, de forma que si uno de los dos servidores sufre una caida el otro toma el control de la ip virtual y puede atender a las peticiones.

Siglas

ACID Atomicity, Consistency, Isolation and Durability. 12

LAMP Linux, Apache, MySql, PHP. 2

Glosario

expresiones regulares secuencias de caracteres que forma un patrón de búsqueda, principalmente utilizada para la búsqueda de patrones de cadenas de caracteres u operaciones de sustituciones . 50

HTTP (*Hyper Text Transfer Protocol*), Es el protocolo que se emplea para las comunicaciónes a través de internet. 3, 4

JSP (*Java Server Pages*), Tecnología similar a PHP, ya que permite crear aplicaciones web dinámicas del lado servidor, excepto que para la máquina virtual de Java. 7, 9

MVC (*Model View Controller*)Patrón de programación que permite separa las tres unidades básicas de un sistema de software, delgando responsabilidades correctamente de forma que las piezas puedan ser reemplazadas. 1, 52

OAuth Es un protocolo de internet que permite la autorización segura de una API. 4

PHP (*PHP HyperText Processor*), Es un lenguaje de programación orientado al desarrollo web dinámico del lado servidor. 6, 9

REST (*Representational State Transfer*), Arquitectura de software que permite desarrollar sistemas distribuidos a través de enlaces de tipo hypermedia, apoyandose en protocolos de apliación como HTTP. 3, 4, 6

SOAP (*Simple Object Access Protocol*), Es un protocolo de comunciación, normalmente a través de HTTP, que permite a dos procesos intercambiar información en formato XML. 2, 3

VIP Virtual IP, es una dirección IP que se asocia a múltiples aplicaciones dentro de un servidor, varios servidores o dominios. . 68

Bibliografía

[1] *http://es.wikipedia.org/wiki/Representational_State_Transfer*

[2] *http://es.wikipedia.org/wiki/JSON*

[3] *http://nginx.org/en/docs/*

[4] *http://es.wikipedia.org/wiki/Simple_Object_Access_Protocol*

[5] Roy Thomas Fielding *Architectural Styles and the Design of Network-based Software Architectures* 2000

[6] Karl Düüna *Analysis of Node.js platform web application security* 2012

[7] *http://nodejs.org/*

[8] *http://docs.mongodb.org/manual/*

[9] *http://aws.amazon.com/es/ec2/*

[10] *http://nginx.org/en/docs/*

[11] Nurzhan Nurseitov *Comparison of JSON and XML Data Interchange Formats: A Case Study* 2003.

[12] Rico Suter *MongoDB An introduction and performance analysis* 2012.

[13] *http://upstart.ubuntu.com/*

[14] *http://mongoosejs.com/docs/guide.html*