Universidad Zaragoza

Trabajo Fin de Grado

# Desarrollo e implantación de un API Web para servir contenido multimedia a dispositivos móviles

## Development and deployment of a Web API to deliver multimedia content to mobile devices


Autor

Eduardo Criado Mascaray

Director

Johan Markus
SOBAKA DEVELOPMENTS AB

Ponente

Francisco Javier Fabra Caro
Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza



Escuela de Ingeniería y Arquitectura
2017

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Eduardo Criado Mascaray,

con nº de DNI 76971132E en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Desarrollo e implantación de un API Web para servir contenido multimedia a dispositivos móviles,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 20 de Noviembre de 2017

Fdo: Eduardo Criado Mascaray

# Agradecimientos

Quisiera agradecer a Johan el haberme dado la oportunidad de llevar a cabo este proyecto y a Javier Fabra por mostrar su interés en él y guiarme durante su realización.
A los profesores de la Universidad de Zaragoza y de la KTH de Estocolmo por los conocimientos que me han transmitido durante mis años de carrera, que me han permitido llegar hasta esta etapa final.
A mis compañeros y amigos con los que he compartido estos años de estudio y buenos momentos.
Por último, a mi familia por hacer todo esto posible.

# Desarrollo e implantación de un API Web para servir contenido multimedia a dispositivos móviles

## Resumen

La música electrónica se ha convertido en un género con una gran popularidad desde que comenzó a tener una enorme influencia en la música popular en las décadas de 1970 y 1980, para finalmente establecerse en la de 1990. Presente en la actualidad en las discotecas y los festivales de música, también cuenta con una amplia audiencia en la Web, a través de páginas como SoundCloud o servicios como Spotify y canales en YouTube. Debido a la gran demanda de contenido relacionado con ella, los pinchadiscos y sus actuaciones, hay plataformas como Boiler Room que ofrecen sesiones de música electrónica que cuentan con cifras de visionado de decenas de millones.

Pocketbeat es una plataforma que ofrece contenido de vídeo y audio de distintos subgéneros como el *techno* o el *house*. Esta se puede acceder desde el navegador, tanto desde ordenadores personales como dispositivos móviles, desde donde proviene la mayor parte de sus usuarios. Tras el análisis de plataformas y aplicaciones similares se han detectado ciertas carencias. Para tratar de cubrirlas y dar un mejor soporte a dichos usuarios se ha decidido invertir en el desarrollo de una aplicación móvil nativa. Esta necesita un API Web que le sirva el contenido de la plataforma, cuyo proyecto se expone en la presente memoria. Los objetivos de este son por tanto el desarrollo e implantación del API y el uso de una infraestructura de integración continua.

Para ello se ha utilizado el *framework* ASP.NET Web API, utilizando Microsoft SQL Server como base de datos y ejecutándose en un servidor Windows con IIS. La infraestructura de integración continua se basa en Atlassian Bitbucket para el control de versiones y TeamCity y Octopus Deploy como servidores de gestión de compilación, validación y despliegue.

Tras pasar una fase de pruebas el API se ha puesto en producción de manera satisfactoria, de forma que se utiliza por la aplicación móvil en actualidad. La infraestructura de integración continua ha resultado de gran utilidad.

# *Índice*

**Glosario** **IX**

**1. Introducción** **1**

1.1. Contexto y motivación . . . . . 1

1.2. Objetivos . . . . . 4

1.3. Estructura de la memoria . . . . . 5

**2. Estado del arte** **7**

2.1. Boiler Room . . . . . 7

2.2. WAV . . . . . 8

2.3. BE-AT.TV . . . . . 9

2.4. Valor añadido en la aplicación de *Pocketbeat* . . . . . 11

**3. Análisis del sistema** **13**

3.1. Requisitos funcionales de la aplicación móvil . . . . . 13

3.2. Análisis de requisitos . . . . . 13

3.2.1. Requisitos funcionales . . . . . 13

3.2.2. Requisitos no funcionales . . . . . 13

3.3. Arquitectura . . . . . 18

3.4. Tecnologías utilizadas . . . . . 19

3.4.1. Amazon Web Services (AWS) . . . . . 19

3.4.2. ASP.NET Web API . . . . . 19

3.4.3. Microsoft SQL Server . . . . . 21

3.4.4. Microsoft IIS . . . . . 21

3.4.5. Wowza Streaming Engine, en servidor Ubuntu . . . . . 21

3.4.6. Git y Atlassian Bitbucket . . . . . 22

3.4.7. TeamCity y Octopus Deploy . . . . . 22

3.4.8. Visual Studio 2017 . . . . . 22

**4. Desarrollo** **25**
4.1. Desarrollo del API RESTful . . . . . 25
4.1.1. Conjunto de recursos . . . . . 26
4.1.2. Modelos de datos . . . . . 26
4.1.3. Separación en controladores . . . . . 30
4.1.4. Seguridad . . . . . 32
4.1.5. Gestión de imágenes . . . . . 33
4.2. Infraestructura de integración continua . . . . . 33
4.2.1. Instancias en Amazon Web Services . . . . . 37

**5. Validación** **39**
5.1. Debug mediante Visual Studio y Postman . . . . . 39
5.2. Conjunto de pruebas unitarias . . . . . 40
5.3. Despliegue de aplicación en servidor de prueba . . . . . 41

**6. Conclusiones** **43**
6.1. Gestión del proyecto . . . . . 43
6.2. Conclusiones . . . . . 44
6.3. Trabajo futuro . . . . . 44
6.4. Opinión personal . . . . . 45

**7. Bibliografía** **47**

**Anexos** **50**

**A. Mapa de la aplicación móvil** **53**

**B. API RESTful de Pocketbeat** **71**

**Lista de Figuras** **73**

**Lista de Tablas** **75**

# *Glosario*

En este capítulo se explican términos importantes que se repiten a lo largo de toda la memoria. Estos son:

- *Application programming interface* (API): conjunto de métodos, procedimientos y herramientas que se proporcionan para construir aplicaciones o programas. En este proyecto se habla de un API Web, es decir, del conjunto de métodos que se han creado para construir aplicaciones (la aplicación móvil), mediante tecnologías propias de la Web.

- *Create, read, update and delete* (CRUD): en español, crear, leer, actualizar y borrar. Son las cuatro operaciones básicas de un sistema de almacenamiento persistente. En el desarrollo de un API RESTful tienen una correlación con los métodos HTTP utilizados.

- *Framework*: en el desarrollo de *software*, abstracción que proporciona un conjunto de conceptos y prácticas genéricas que se pueden adaptar para un problema en particular.

- Integración continua: práctica de programación que promueve el lanzamiento de código con nuevas funcionalidades con alta frecuencia, más de una vez al día, con automatización en los procesos del lanzamiento. En este proyecto se adopta el concepto para referirse a la arquitectura que permite el despliegue del API.

- Modelo-vista-controlador (MVC): patrón de diseño *software* tradicionalmente utilizado en aplicaciones que incluyen una interfaz de usuario. Organiza la aplicación en tres tipos de componentes (modelos, vistas y controladores) con funciones distintas y que se comunican entre sí. Los modelos contienen la lógica de negocio y datos, las vistas presentan la información y los controladores reciben la entrada del usuario que transforma en acciones para los modelos o las vistas. El *framework* utilizado para este proyecto (ASP.NET Web API) cuenta con una estructura inicial basada en este patrón pero sin contar con las vistas, que serían propias de por ejemplo una página Web.

– *Representational State Transfer* (REST): arquitectura *software* utilizada en sistemas distribuidos. Define un conjunto de restricciones y prácticas que generan ventajas como escalabilidad o su consumo independiente de la plataforma del cliente. Un API Web que sigue la arquitectura REST se conoce como RESTful. Se explica en mayor profundidad en la sección 4.1.

– Sistema de control de versiones: gestiona los cambios que se producen en los ficheros que componen un programa informático (generalmente ficheros de código fuente, pero no únicamente). Hay de distintos tipos: locales, que siguen un modelo cliente-servidor, distribuidos, etc. En este proyecto se utiliza un sistema distribuido, detallado en la subsección 3.4.6.

# 1. *Introducción*

En este capítulo se describe el contexto en el cual el proyecto se ha desarrollado y las razones de su existencia. Después se enumeran los objetivos que se pretenden alcanzar con el mismo, y por último la estructura de la presente memoria.

## 1.1. Contexto y motivación

La empresa SOBAKA DEVELOPMENTS AB[1], situada en Suecia, desarrolla proyectos tecnológicos, como aplicaciones móviles o páginas Web, y ofrece servicios de publicidad en Internet. Uno de estos proyectos, y el que más importancia tiene actualmente, es *Pocketbeat*[2]. *Pocketbeat* es una plataforma de contenido de vídeo y audio centrada en distintos subgéneros de la música electrónica. Intenta ofrecer una experiencia lo más cercana posible a los aficionados de estos géneros de fiestas y actuaciones de *DJs*. El contenido más frecuente es de tipo vídeo, con una duración de entre una hora a dos, que corresponde al tiempo medio de las sesiones de pinchadiscos.

Esta plataforma se puede acceder desde el navegador, tanto desde ordenadores personales como dispositivos móviles, ya que cuenta con un diseño adaptativo. En las figuras 1.1 y 1.2 se observa el aspecto de la página Web desde el navegador de un ordenador personal. En las figuras 1.3 y 1.4 el mismo pero desde un dispositivo móvil.

La plataforma cuenta con un gran porcentaje de usuarios que principalmente acceden a ella utilizando dispositivos móviles[1]. A pesar de los grandes avances en diseño de aplicaciones Web para móviles, el desarrollo de aplicaciones nativas sigue ofreciendo ciertas ventajas, que sin embargo conllevan un mayor coste de desarrollo. Algunas de estas ventajas son:

- Contenido *offline* disponible: aunque no se ha incluido en esta iteración de desarrollo, es una de las funcionalidades que se prevé que atraiga a un número importante de usuarios.

- Acceso a mayores funcionalidades y sensores del dispositivo: de una forma más

[1]Mediante la herramienta *Google Analytics* se comprobó que desde el 1 de agosto al 1 de Septiembre de 2017, el porcentaje de acceso desde dispositivos móviles era del 64 %.

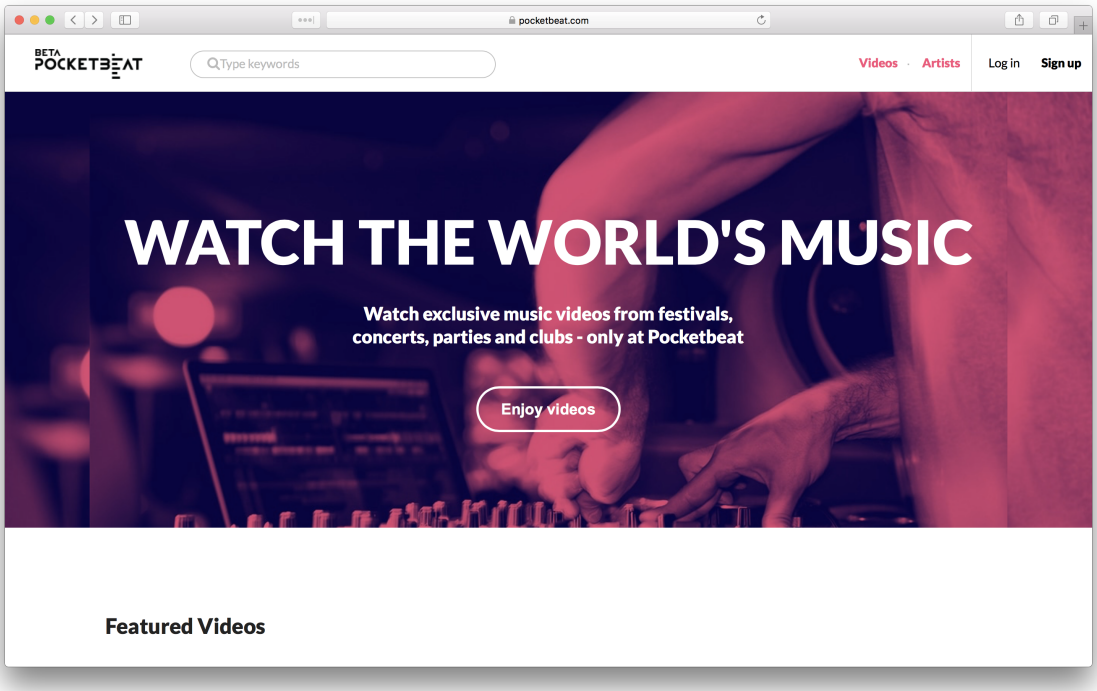


Figura 1.1: Captura de pantalla de la página de Pocketbeat accedida desde un ordenador de escritorio.

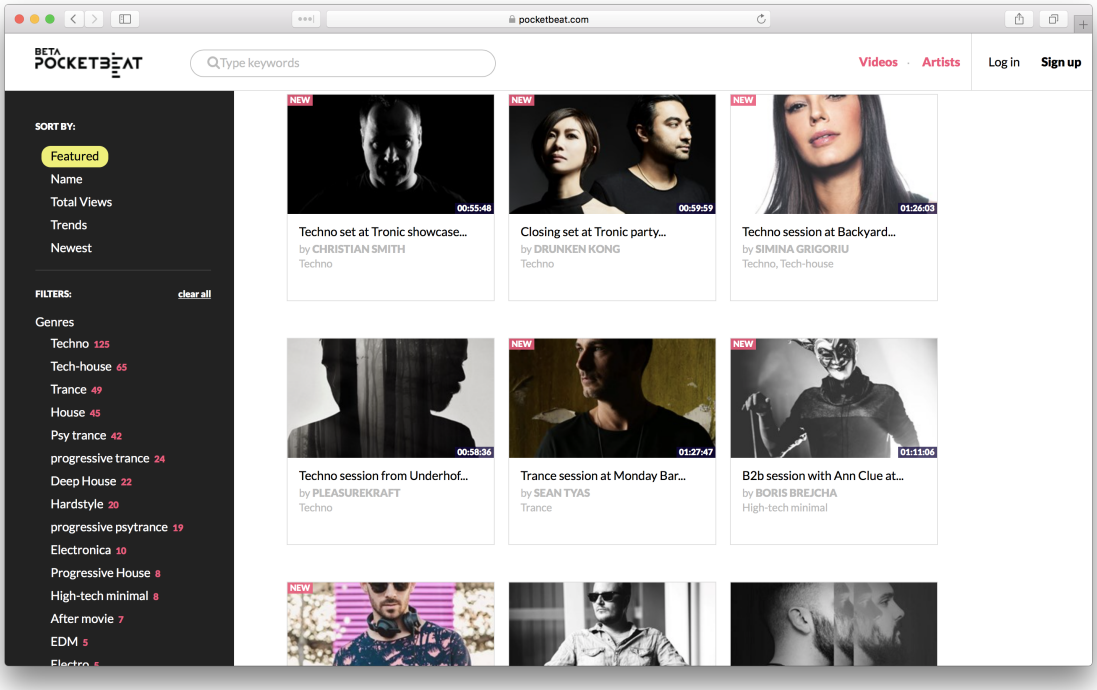


Figura 1.2: Captura de pantalla de la página de Pocketbeat accedida desde un ordenador de escritorio. Vista de vídeos.

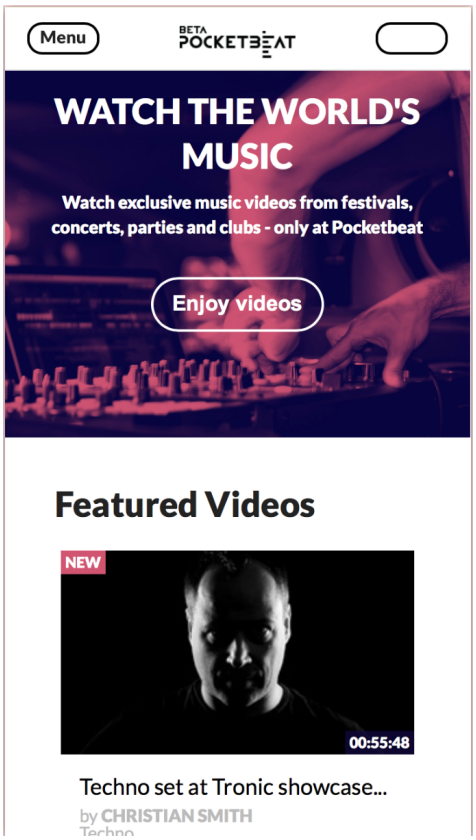


Figura 1.3: Vista inicial.

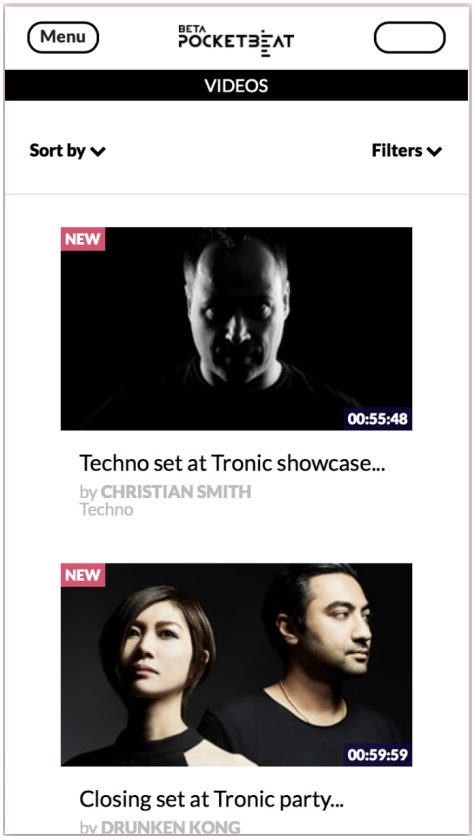


Figura 1.4: Vista de vídeos.

sencilla poder acceder a la cámara o el GPS. Las librerías nativas, como la de reproducción de vídeo en iOS, ofrecen una alta libertad de configuración y un rendimiento superior.

– Visibilidad en el dispositivo del usuario: si la aplicación se encuentra instalada y el usuario observa el icono mientras realiza otras tareas es más probable que la utilice con mayor frecuencia, frente a quizás encontrarse como un marcador en el navegador Web del teléfono.

– Aparición en la tienda de aplicaciones de la plataforma: mayor confianza por parte del usuario al utilizar la plataforma, al haberse aprobado según los criterios de la tienda. Se trata de una aplicación nativa y no de una aplicación híbrida donde se ha encapsulado la página Web en un una vista de navegador.

– Interfaz que aproveche los principios de diseño y recursos, como transiciones o botones, que proporciona la plataforma.

Por estas razones se decicidió desarrollar una aplicación móvil para acceder a la plataforma.

El API Web desarrollado pretende proporcionar la información necesaria a la

aplicación móvil mediante el uso de un API Web. Dado que el API con el que cuenta la aplicación Web está fuertemente ligado a las vistas[2], no se ha podido reutilizar para la aplicación móvil. La aplicación móvil se encuentra actualmente en desarrollo. Aunque se está desarrollando mediante Xamarin[3] para cubrir tanto iOS como Android, el lanzamiento inicial se va a realizar centrándose en iOS, que cuenta con una mayor cuota de mercado en Suecia. Xamarin es una tecnología que permite desarrollar aplicaciones móviles nativas en C# para distintas plataformas, de forma que compartan la lógica de negocio, pero difieran en la parte de la interfaz.

La aplicación se encuentra en fase de *beta testing* y disponible mediante *TestFlight*[4]. *TestFlight* es un servicio que ofrece *Apple* para los desarrolladores de aplicaciones móviles para distribuirlas, de forma que se pueda invitar a otras personas para que las prueben.

Otro de los aspectos que se decidieron para este proyecto fue el uso de una infraestructura de integración continua que permitiera la automatización de compilado y despliegue del proyecto.

Con este proyecto, teniendo en cuenta con que forma parte del desarrollo de la aplicación móvil, se pretende aumentar el número de usuarios de la plataforma, y cuando el número de usuarios sea considerable dar pasos hacia la monetización de esta. Esto se ha planeado de distintas formas. Una opción es adoptar un modelo *freemium*, donde parte del contenido y funcionalidades sean gratis para todos los usuarios y se reserven ciertas para usuarios que paguen por la plataforma. Otra opción que se está considerando es la búsqueda de inversión por parte de otras plataformas como Boiler Room[5], que cuentan con un gran número de usuarios, pero con una página Web o aplicación inferior.

## 1.2. Objetivos

Los objetivos de este proyecto son por tanto:

– Desarrollar un API que permita el acceso al contenido de la plataforma desde una aplicación móvil, independientemente de su sistema operativo. Aunque el desarrollo del API se haya realizado en concreto para ser utilizada por una aplicación móvil, puede ser utilizada en el futuro para ofrecer los datos a terceros, que puedan utilizar la información para sus propias aplicaciones, o para un rediseño de la aplicación Web. Un ejemplo de rediseño sería cambiar a una aplicación *front-end* con alguna librería hecha en JavaScript, como React o Angular, que consumiera los datos del API.

[2]La aplicación Web sigue el patrón Modelo-vista-controlador.

– Crear una estructura de integración continua. En la primera fase de desarrollo de la aplicación los cambios serán frecuentes y es importante reducir el tiempo de despliegue lo máximo posible. Dado el uso de integración continua en el mantenimiento y desarrollo de la página Web de *Pocketbeat* se puede aprovechar la tecnología para crear una infraestructura similar para el API.

– Desplegar el API primero en un entorno de pruebas y más adelante en producción. El entorno de pruebas deberá ser lo más parecido posible al entorno de producción. Se utilizará la infraestructura mencionada en el objetivo anterior para este mismo.

## 1.3. Estructura de la memoria

La memoria del trabajo está formada por los siguientes capítulos y anexos:

– Glosario: definición de términos utilizados a lo largo de la memoria.

– Capítulo 1, Introducción: se introduce el proyecto, detallando los motivos por los que se ha llevado a cabo y su contexto, los objetivos y la estructura de la memoria.

– Capítulo 2, Estado del arte: se enumeran aplicaciones con un propósito y contenido similar al del de *Pocketbeat* y se justifica qué ofrece esta frente a las demás.

– Capítulo 3, Análisis del sistema: contiene una descripción de los objetivos, de una manera más minuciosa que en la introducción. A continuación se enumeran los requisitos de la aplicación móvil, y a partir de estos los del API Web, tanto funcionales como no funcionales. Tras esto se pasa a explicar la arquitectura que compone el sistema y las tecnologías utilizadas junto a su justificación.

– Capítulo 4, Desarrollo: se describe el proceso de desarrollo del API, con todas las decisiones tomadas, además de la infraestructura para el despliegue de esta.

– Capítulo 5, Validación: contiene la explicación de las pruebas para asegurar el correcto funcionamiento del API; primero las no formales y las herramientas utilizadas y luego las formales en forma de *tests* unitarios.

– Capítulo 6, Conclusiones: se describe la gestión del proyecto, sus conclusiones, el trabajo futuro a realizar y la opinión personal sobre el mismo.

– Capítulo 7, Bibliografía: contiene el conjunto de referencias consultadas.

– Anexo A, Mapa de la aplicación móvil: conjunto de pantallas de la aplicación e interacción entre ellas.

– Anexo B, API RESTful de Pocketbeat: se detallan las operaciones por recurso del API.

# 2. *Estado del arte*

En este capítulo se describen aplicaciones móviles que ofrecen un contenido similar, se analizan sus puntos fuertes y débiles y qué ofrece *Pocketbeat* frente a ellas. Sólo se incluyen aplicaciones para iOS, que es la plataforma elegida para el lanzamiento de la aplicación de *Pocketbeat* por ser la más utilizada en Suecia.

## 2.1. Boiler Room

Boiler Room es la plataforma más parecida, tanto en propósito como en contenido. Contiene el mayor número de vídeos sobre fiestas y conciertos. Aunque originalmente centrada en la música electrónica, se ha expandido a otros géneros como el hip hop o el jazz. Su canal de YouTube[6] cuenta con cerca de 1,4 millones de subscriptores y aproximadamente 5500 vídeos. Su página Web ofrece el mismo contenido que el canal de YouTube pero mejor organizado, ya que cuenta con 4 canales o *channels*, donde el contenido está organizado por géneros similares (música electrónica británica como *garage* y *grime* en el primero, *house* y *techno* en el segundo, etcétera). Además las mismas sesiones están disponibles en formato audio a través de SoundCloud.

Cuenta además con una aplicación móvil[7]. La aplicación móvil ofrece las mismas funcionalidades que la página Web y además permite la descarga de las sesiones al dispositivo en formato audio. Las descargas de audio se pueden acceder después a través de una de las pestañas inferiores, para reproducir o eliminar. En la pestaña *Explore* se puede filtrar el contenido por géneros, formatos (sesiones en discotecas, en estudio, documentales, etc.) y series. Se ha incluido una captura de pantalla en la figura 2.1 para observar su diseño.

Los puntos más fuertes de la aplicación son la cantidad de contenido (aunque no sea una funcionalidad, un usuario se siente atraído por esto), la organización de la música en canales, la reproducción tanto de vídeo como audio y la posibilidad de descargar sesiones para no tener que utilizar la tarifa de datos, además de una interfaz sencilla y pulida. Por el contrario, flaquea en cuanto a que no se puede buscar por artista, no se pueden realizar comentarios ni guardar sesiones, o artistas, como favoritos. En resumen la aplicación resulta demasiado sencilla, podría ofrecer más funcionalidades y


Figura 2.1: Captura de pantalla de la aplicación de Boiler Room, versión 3.0.9.

mantener igualmente un diseño minimalista.

## 2.2. WAV

WAV[8] no se parece tanto en contenido; aunque también se centra en música electrónica, ofrece en mayor medida contenido sobre la cultura *hip hop*. Además, no está tan enfocada a sesiones completas de actuaciones, sino también videoclips, entrevistas o resúmenes de conciertos. Las funcionalidades sin embargo son similares, y en cierta medida muy influenciadas por Boiler Room. Cuenta con una organización del contenido en canales igual que Boiler Room, pero estos son más bien canales promocionados, donde revistas como Fader muestran su contenido. Sí que permite la búsqueda por artista y un perfil de usuario. El perfil de usuario ofrece la posibilidad de almacenar artistas favoritos y subscripción a canales y series (entendidas como vídeos con un contenido y formato parecido), además de la realización de comentarios en los vídeos. En la captura de pantalla mostrada en la figura 2.2 se puede observar que el diseño es muy similar al de la aplicación de Boiler Room, en la figura 2.1.

A pesar de contar con mayor funcionalidades que la aplicación de Boiler Room, el contenido es mucho más limitado y no se permite su descarga ni la opción de cambiar

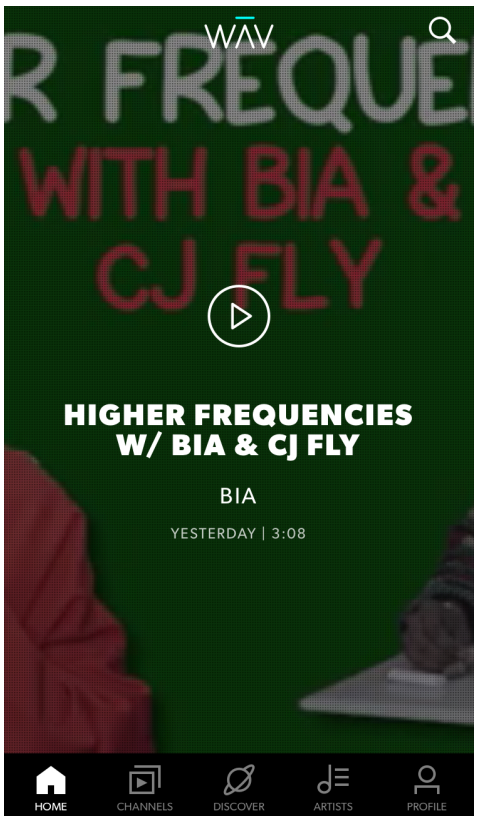


Figura 2.2: Captura de pantalla de la aplicación WAV, versión 1.3.

entre audio y vídeo.

## 2.3. BE-AT.TV

BE-AT.TV[9] se trata probablemente de la aplicación[10] con un propósito más parecido al de *Pocketbeat*, ya que está únicamente enfocada a sesiones de pinchadiscos de música electrónica. Sin embargo, se ha valorado si incluirla o no en este capítulo debido a que la última versión (1.4) de la aplicación es de hace dos años. Esto es palpable en el diseño, bastante anticuado. Se puede observar el aspecto de la aplicación en las figuras 2.3 y 2.4.

Sin embargo incluye muchas funcionalidades, como explorar por artistas, cuentas de usuario, se puede comentar en las sesiones, compartirlas. También cuenta con un calendario que contiene información de eventos planificados. Resulta extraño que no sea haya continuado su desarrollo porque el contenido de sesiones que muestra es actual.

El reproductor de vídeo por otra parte deja mucho por desear, ya que sólo permite el pausado de la sesión, no hay ningún tipo de control para dirigirse a un punto en el tiempo anterior o siguiente ni una barra que indique el progreso.

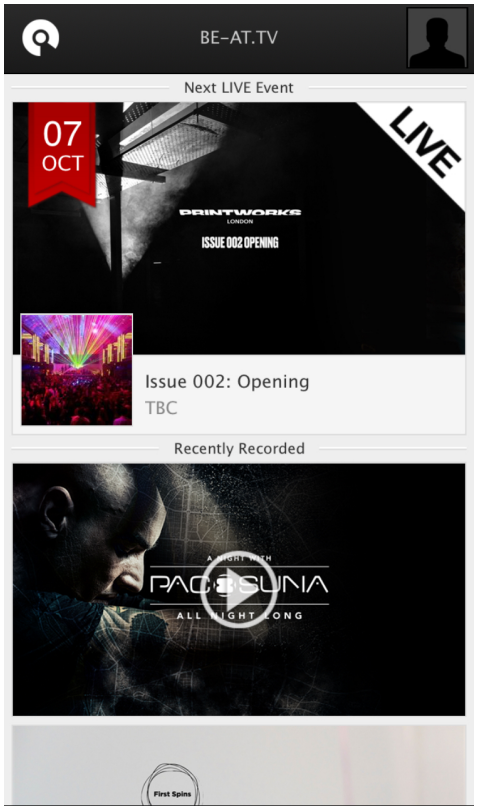


Figura 2.3: Captura de pantalla de la aplicación BE-AT.TV, pantalla inicial. Versión 1.4.

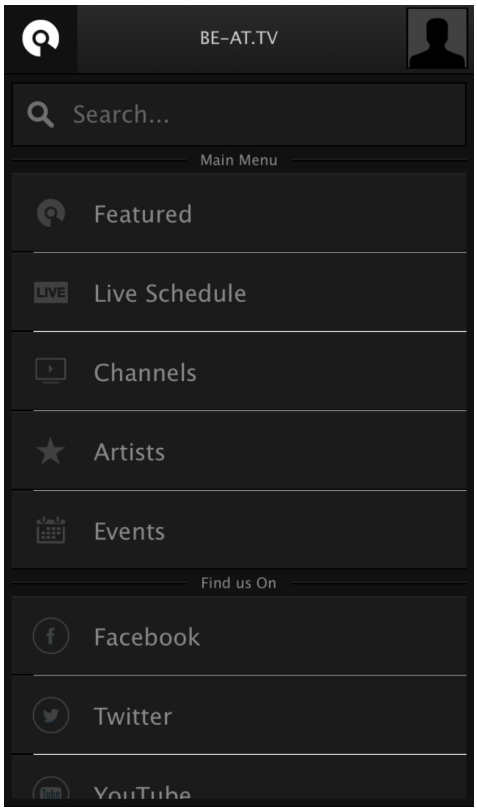


Figura 2.4: Captura de pantalla de la aplicación BE-AT.TV, menú lateral. Versión 1.4.

## 2.4. Valor añadido en la aplicación de *Pocketbeat*

*Pocketbeat* trata de cubrir un número de funcionalidades que otras aplicaciones similares del mercado carecen. La aplicación que se está desarrollando ofrece un contenido de alta calidad, aunque pequeño si se compara a Boiler Room (aproximadamente quinientas sesiones frente a las más de cinco mil) pero suficientemente variado para ser atractivo a los usuarios. Ofrece funcionalidades como el acceso mediante una cuenta de usuario, guardado de artistas y sesiones favoritas, realizado de comentarios, cambio entre audio y vídeo, y en un futuro cercano contenido *offline*. Es posible explorar por artistas, filtrando por inicial, y la exploración de sesiones permite filtrado por género y orden según contenido destacado, las sesiones más nuevas y más antiguas, "de moda" y las más vistas. Todo esto manteniendo una interfaz actual, minimalista y usable.

# 3. *Análisis del sistema*

Tras la introducción, en este capítulo se va a dar una explicación del análisis de requisitos del API, partiendo de los requisitos de la aplicación móvil. Además se va a proponer la arquitectura del sistema. Por último, se justificarán las tecnologías utilizadas para su desarrollo. Los objetivos ya se han explicado en la sección 1.2.

## 3.1. Requisitos funcionales de la aplicación móvil

Los requisitos funcionales de la aplicación móvil aparecen en la tabla 3.1.

Por sesión se entiende una grabación de una actuación en directo de un pinchadiscos, ya sea en formato vídeo o sólo audio. La sesión que más "de moda" está es la que acumula más visionados durante el día de consulta. El contenido destacado se elige desde un panel de control de administrador de la aplicación Web, no se trata de ningún algoritmo.

En la figura 3.1 se puede observar un ejemplo de pantalla de la aplicación móvil, simplemente, para exponer el tipo de información que los dispositivos móviles van a tratar. En el anexo A se encuentran todas las pantallas de la aplicación.

## 3.2. Análisis de requisitos

Se han visto los requisitos de la aplicación móvil, y por tanto se pueden derivar de ellos los requisitos que el *API* debe tener. Se presentan a continuación.

### 3.2.1. Requisitos funcionales

Los requisitos funcionales aparecen en la tabla 3.2.

La información de métodos y URIs específicas para el funcionamiento de la aplicación están especificadas en el capítulo 4.

### 3.2.2. Requisitos no funcionales

Los requisitos no funcionales aparecen en la tabla 3.3.

| ***Código*** | ***Descripción*** |
|---|---|
| RF1 | El usuario de la aplicación tiene que ser capaz de identificarse, usando usuario y contraseña. |
| RF2 | El usuario debe poder registrarse, introduciendo nombre, correo electrónico, contraseña y una imagen de perfil. |
| RF3 | El usuario tiene que poder recuperar la contraseña, solicitando que se le reenvíe un enlace para crear una nueva al correo electrónico. |
| RF4 | El usuario podrá explorar una serie de sesiones de artistas y filtrarlas según género musical y por más visto, "de moda", destacado, más nuevo y más antiguo. |
| RF5 | El usuario podrá reproducir una sesión y cambiar entre audio y video. |
| RF6 | El usuario podrá guardar esa sesión, que pasará a formar parte de una lista con sesiones guardadas. |
| RF7 | El usuario podrá compartir la sesión, en distintas redes sociales u obteniendo un enlace. |
| RF8 | El usuario podrá publicar un comentario en una sesión. |
| RF9 | Al reproducir una sesión, se mostrará información sobre sesiones del mismo artista y sesiones relacionadas con la que está reproduciendo. |
| RF10 | El usuario podrá explorar los artistas de la plataforma y filtrar según sus iniciales. |
| RF11 | El usuario podrá seguir a un artista. Al seguir a un artista el artista aparecerá en un menú que contenga los artistas seguidos. |
| RF12 | El usuario podrá realizar búsquedas de texto cuyos resultados podrán ser tanto de artistas como de sesiones, que aparecerán conforme escriba. |
| RF13 | El usuario podrá acceder a un menú de ajustes en el cual podrá editar su nombre, país de residencia, etc. |

Tabla 3.1: Requisitos funcionales de la aplicación móvil.

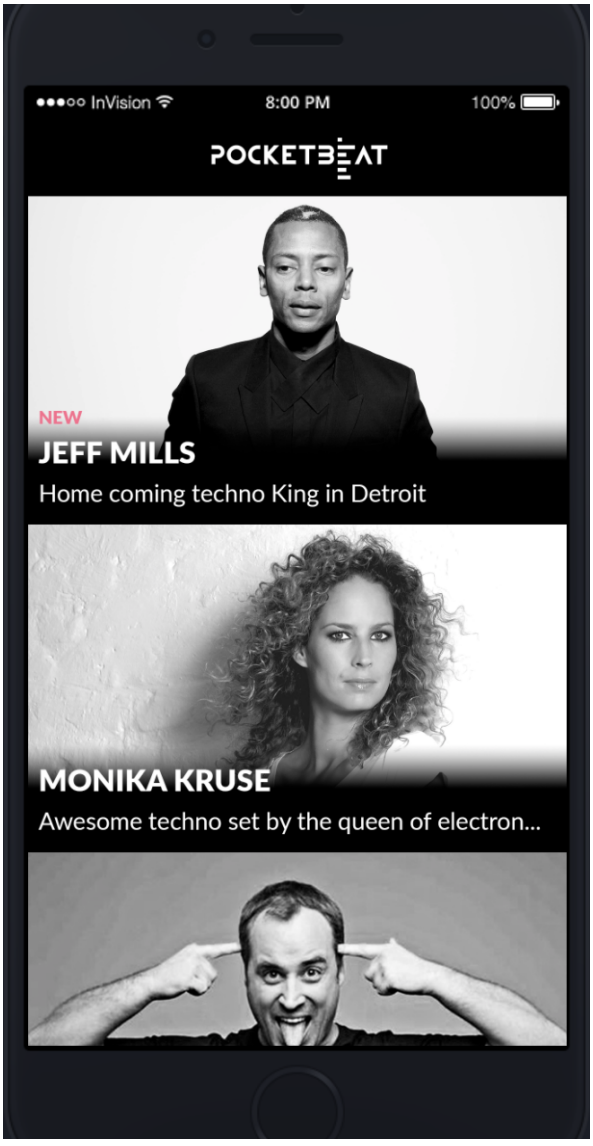


Figura 3.1: Pantalla principal aplicación.

| ***Código*** | ***Descripción*** |
|---|---|
| RF1 | El sistema debe responder a cada petición de acceso a un recurso con el recurso especificado. |
| RF2 | El sistema debe responder con un mensaje de error en el caso de que el recurso especificado sea inexistente o el usuario que realiza la petición no tenga acceso a él. Para esto el sistema hará uso de los códigos de error establecidos en HTTP. |
| RF3 | El sistema debe aceptar únicamente métodos HTTP estándar del repertorio CRUD (GET, POST, PUT y DELETE). |
| RF4 | El sistema debe validar si una petición que afecta a datos únicamente accesibles por un usuario proviene de dicho usuario. El mecanismo elegido para realizar la autenticación se decidirá más adelante. |
| RF5 | El sistema debe contener un recurso que corresponda a sesiones, filtrando según los parámetros: género, más visionados, "de moda", destacado, más nuevo y más antiguo. Además para obtener una sesión individual se puede utilizar su identificador. Se podrá solicitar información sobre sesiones del mismo artista y sesiones relacionadas. |
| RF6 | El sistema debe contener un recurso que corresponda a artistas, con posibilidad de filtrar por inicial. También se podrá obtener artistas individuales con su identificador. Debe permitir añadir como seguido el artista por parte de un usuario. Se podrá obtener sesiones del artista. |
| RF7 | El sistema debe contener un recurso que corresponda al usuario, que permita modificar sus campos como correo electrónico, nombre o ciudad. |
| RF8 | El sistema debe contener un recurso que corresponda a comentarios de una sesión. Tiene que ofrecer la funcionalidad de leerlos y crear nuevos. |
| RF9 | El sistema debe aceptar una cadena de texto como búsqueda, que devuelva resultados tanto de artistas como sesiones. |

Tabla 3.2: Requisitos funcionales del API

| ***Código*** | ***Descripción*** |
|---|---|
| RNF1 | La comunicación entre cliente y servidor debe de ser cifrada, mediante HTTPS. |
| RNF2 | Los modelos de sesión contendrán la siguiente información:<br>– Identificador.<br>– Título.<br>– Descripción.<br>– Enlace de visionado.<br>– Imagen en miniatura.<br>– Identificadores de/de los artista/s.<br>– Nombre de/de los artista/s. |
| RNF3 | Los modelos de artista contendrán la siguiente información:<br>– Identificador.<br>– Nombre.<br>– Biografía.<br>– Imagen. |
| RNF4 | Los modelos de usuario contendrán la siguiente información:<br>– Identificador/Correo electrónico.<br>– Nombre.<br>– Contraseña.<br>– País.<br>– Ciudad.<br>– Teléfono móvil.<br>– Imagen. |
| RNF5 | Los modelos de comentario contendrán la siguiente información:<br>– Identificador del autor.<br>– Nombre del autor.<br>– Texto.<br>– Fecha. |
| RNF6 | Las respuestas del sistema deben de estar de estar en formato JSON. |

Tabla 3.3: Requisitos no funcionales del API

## 3.3. Arquitectura

En esta sección se describe la arquitectura. La arquitectura se ha dividido en dos, dado que primero se explica la arquitectura que corresponde al funcionamiento de la aplicación, y después la arquitectura de la infraestructura de integración continua. El diagrama de la figura 3.2 corresponde a la arquitectura propuesta para el funcionamiento de la aplicación.

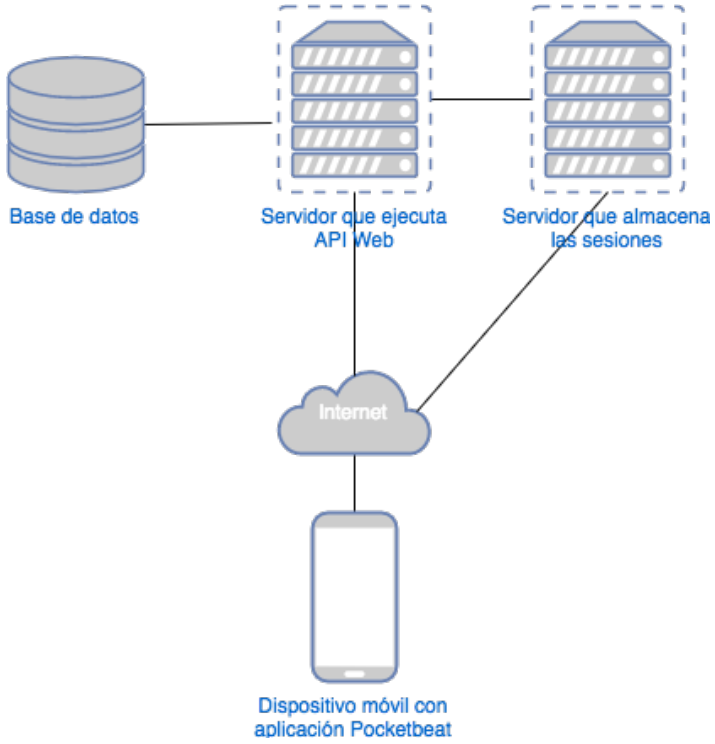


Figura 3.2: Arquitectura del sistema.

El diagrama de la figura 3.3 corresponde a la arquitectura de integración continua. A continuación se explica cada elemento de los diagramas anteriores:

- Base de datos: contiene los datos de la plataforma, de los cuales se extraerán los necesarios dados los requisitos anteriores.
- Servidor con API: es el servidor que aloja la aplicación que ofrece la funcionalidad de API.
- Servidor que almacena las sesiones: es el servidor que almacena los ficheros con las sesiones, bien en formato vídeo o audio.
- Servidor con repositorio de código remoto: permite el trabajo en equipo con otros desarrolladores y a la vez cumple la función de ser una copia se seguridad de la aplicación en todo momento.

– Servidor que compila y despliega paquetes: facilita el despliegue y ofrece ventajas como establecer procesos rutinarios antes y después de la operación o gestión de versiones del *software*.

## 3.4. Tecnologías utilizadas

Los diagramas de arquitectura con las distintas tecnologías son los siguientes: la figura 3.4 corresponde al de la aplicación y la figura 3.5 al de la infraestructura de despliegue.

En esta sección se enumera el conjunto de tecnologías utilizadas y su razón. La tecnología de mayor importancia para el desarrollo del proyecto es sin duda ASP.NET[11], un *framework* enfocado a la Web desarrollado por Microsoft. Utiliza el lenguaje de programación C#. En concreto se ha utilizado el *framework* ASP.NET Web API, que está enfocado, como su nombre indica, a la creación de APIs Web. Dado que se trata de un *framework* específicamente orientado a este tipo de aplicaciones contiene muchas utilidades que hacen el desarrollo más accesible.

### 3.4.1. Amazon Web Services (AWS)

Se ha elegido AWS[12] como proveedor de servicios en la "nube". Ofrece soluciones para todos los elementos que componen la arquitectura, tanto de la aplicación como de la infraestructura de integración continua. Para la base de datos ofrece una solución llamada Relational Database Service (RDS)[13]. En el caso de servidores, ofrece Elastic Compute Cloud (EC2)[14], que contiene una gran variedad de máquinas virtuales, con hardware adaptado para distintas situaciones, como máquinas con necesidad de alta capacidad de memoria o alta velocidad de computación. En este caso, la familia de máquinas de propósito general será suficiente para las necesidades de la aplicación. Por esta versatilidad de productos se ha elegido este proveedor de servicios *cloud*.

### 3.4.2. ASP.NET Web API

Como ya se ha nombrado, *framework* especializado en el desarrollo de APIs Web. Debido a que la mayor parte del código del proyecto *Pocketbeat* se encuentra en el ecosistema de Microsoft (servidor IIS[15], ASP.NET MVC[16] para la página Web, Microsoft SQL Server[17] para la base de datos) tiene sentido mantener la misma tecnología para la página web.

Este *framework* cuenta además con un gran número de paquetes de libre uso, accesibles mediante NuGet[18]. NuGet es un gestor de paquetes para la plataforma

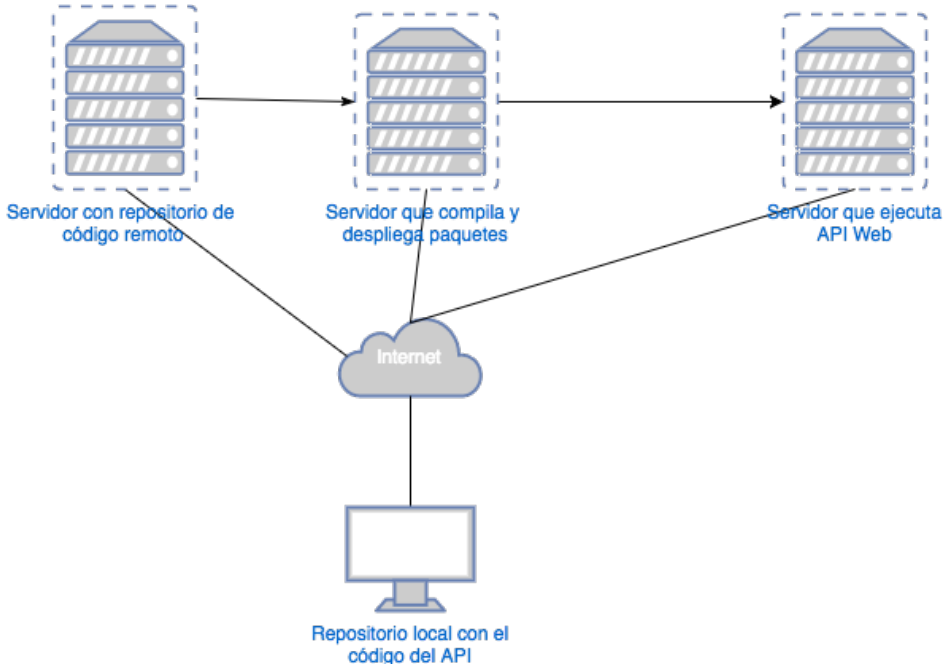


Figura 3.3: Arquitectura del sistema de integración continua.

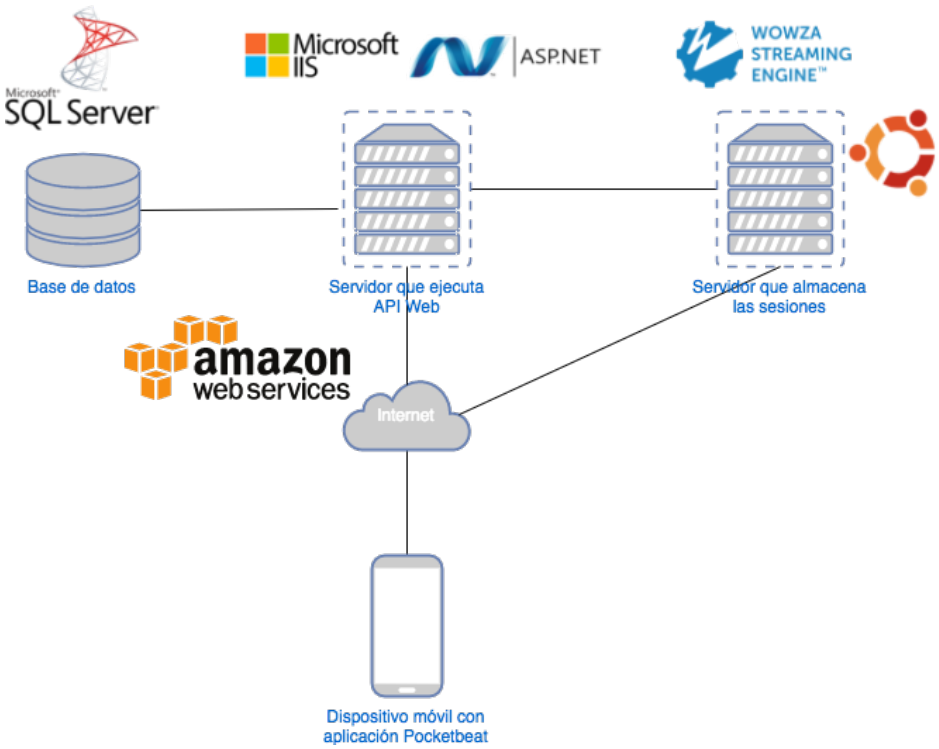


Figura 3.4: Arquitectura del sistema con iconos de las tecnologías utilizadas.

.NET, cuenta con más de 90.000 paquetes distintos. Algunos de los que se van a utilizar en concreto para este proyecto son:

- Entity Framework[19]: provee un mapeo objeto-relacional. De esta manera no hay necesidad de preocuparse de la creación de tablas en SQL y las migraciones cuando se cambian los distintos modelos son sencillas gracias a los distintos comandos que ofrece.

- Newtonsoft.Json[20]: permite la serialización y deserialización entre objetos de ASP.NET y objetos JSON, que es la notación que nuestra API Web utiliza.

- AutoMapper[21]: ofrece un mapeo objeto a objeto. La tarea de mapeo entre objetos resulta monótona, y si aparece en distintas localizaciones del código, repetitiva. AutoMapper ofrece mapeo automático de campos de objetos con mismos nombres, y una sintaxis muy sencilla para los campos que no tienen el mismo nombre en distintos objetos o cuando es necesario algún tipo de transformación.

- log4net[22]: permite escribir la salida del registro a ficheros o una base de datos entre otros y soporta distintos niveles de registro (*Debug*, *Info*, *Warn*, etc.).

### 3.4.3. Microsoft SQL Server

Para la base de datos se ha seleccionado Microsoft SQL Server. Se trata de una estancia de la base de datos desplegada utilizando el producto RDS[13], que ya se ha nombrado. Este mismo sistema se encuentra en uso en la aplicación Web de *Pocketbeat*, con un funcionamiento satisfactorio. En ningún momento es necesario el uso de SQL para la creación de tablas o consultas. La creación de tablas se realiza con el *Entity Framework* y la consulta con el componente LINQ de ASP.NET.

### 3.4.4. Microsoft IIS

Microsoft IIS[15] es un servidor Web para la familia de sistemas operativos Windows. Mediante el *software* IIS Manager se puede gestionar el servidor Web con una interfaz sencilla y manejable. De la misma forma que con la base de datos, este software está en uso para la aplicación Web, y se mantiene para la API.

### 3.4.5. Wowza Streaming Engine, en servidor Ubuntu

Wowza Streaming Engine[23] software se encarga de proveer transmisión de contenido multimedia, tanto bajo demanda como en directo. Este servidor almacena

por tanto los ficheros que contienen las sesiones y genera URLs para poder acceder a ellos. Abstrae la comunicación entre este servidor y el reproductor de vídeos de la página Web o el de la aplicación móvil.

### 3.4.6. Git y Atlassian Bitbucket

Para el control de versiones del código se utiliza Git[24], de forma local, y de forma remota utilizando Atlassian Bitbucket[25]. Mediante el sistema de ramas se mantienen distintas versiones del código para una aplicación en pruebas y para la aplicación en producción. La estructura de pruebas se explica en el capítulo 5.

### 3.4.7. TeamCity y Octopus Deploy

TeamCity[26] es un servidor de gestión de compilación e integración continua desarrollado por la compañía JetBrains. Da soporte a aplicaciones basadas en Java, ASP.NET y Ruby. Se puede configurar para detectar cambios en un repositorio remoto, de forma que cuando el código sea diferente se compile automáticamente. Octopus Deploy es un servidor para el despliegue y gestión de versiones. Soporta, entre otros, el sistema de paquetes NuGet.

Por tanto, el proceso total será: TeamCity se encarga de controlar cambios en el repositorio con el código, compilar las distintas aplicaciones, realizar las pruebas unitarias y enviar los paquetes al servidor Octopus. Desde el servidor Octopus se puede crear un nuevo lanzamiento, asignando un número de versión y descripción para destacar las novedades que incluye, por ejemplo. En la figura 3.6 se puede observar un diagrama con las partes que corresponden a cada uno.

### 3.4.8. Visual Studio 2017

Por último, aunque no se haya nombrado en el diagrama de tecnologías en la figura 3.5, el desarrollo de la aplicación se ha realizado en su integridad utilizando el entorno de trabajo de Microsoft Visual Studio[28], que incluye un gran número de herramientas. A continuación se nombran las más útiles:

- Gestión de paquetes NuGet: permite instalar, actualizar, eliminar paquetes.

- Explorar la base de datos mediante el *Server explorer*: permite no sólo explorar bases de datos, tanto locales como remotas, sino servidores y máquinas en Azure (aunque esto no se haya utilizado, dado que las máquinas virtuales se encuentran alojadas utilizando Amazon Web Services).

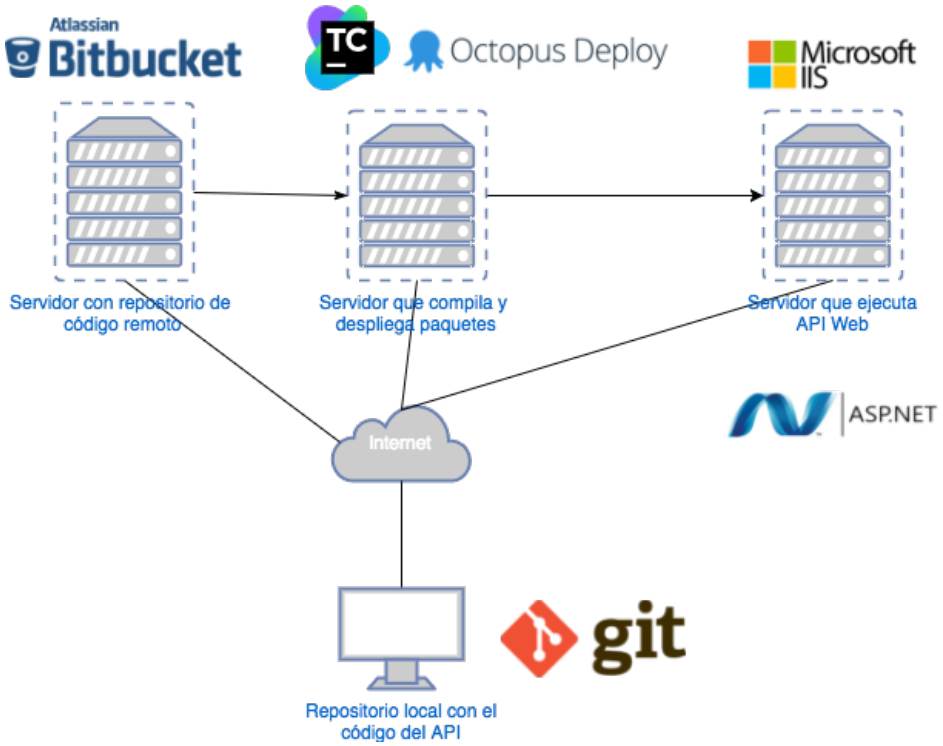


Figura 3.5: Arquitectura de la infraestructura de integración continua del sistema con iconos de las tecnologías utilizadas.

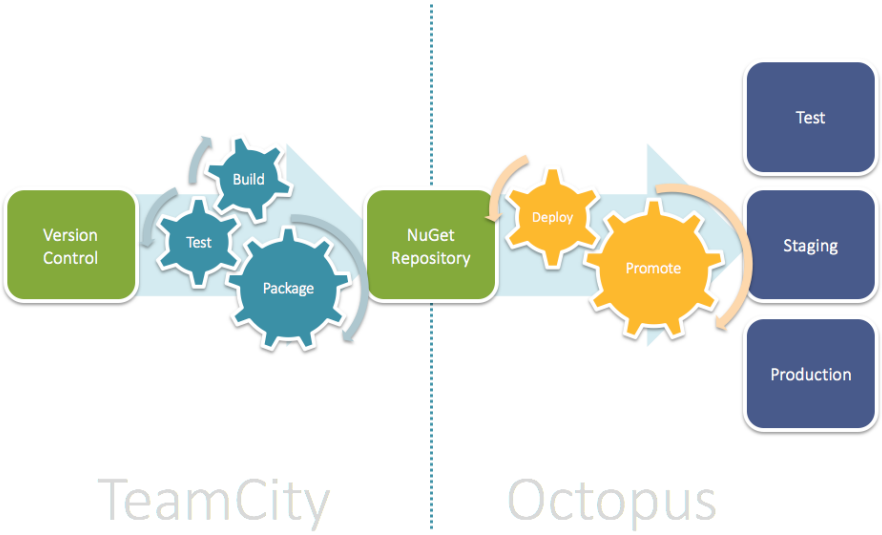


Figura 3.6: Diagrama del proceso, con los diferentes procesos que realiza TeamCity y Octopus. Obtenido de un artículo[27] oficial de JetBrains.

– *Debug mode*: el modo de depuración tiene lo que se espera; gestión de *breakpoints*, análisis de las variables, pila de llamadas, gestión de memoria RAM, excepciones, eventos, etc. Es muy completo.

– Integración con Git: gestión de ramas, *commits*, sincronización con el servidor remoto, *merges* y más operaciones, sin necesidad de utilizar la línea de comandos.

# 4. *Desarrollo*

En este capítulo se describe el proceso de desarrollo del API, con todas las decisiones tomadas. A continuación se describe en más detalle la infraestructura desarrollada para el despliegue en producción de la aplicación.

## 4.1. Desarrollo del API RESTful

En esta primera sección se explica el conjunto de decisiones tomadas a partir del análisis de requisitos del API Web. Aunque ya se haya definido REST previamente, es importante explicar algunos conceptos.

La información está representada por recursos (*resources*), que tienen una identificador asociado; utilizando REST sobre HTTP el identificador es de tipo URI. El estado, tanto actual como el deseado al realizar una actualización, de un recurso se documenta a través del concepto de representaciones (*representations*). Las representaciones se codifican en nuestro caso utilizando el formato JSON. Como ya se ha nombrado en el glosario, REST es un modelo cliente-servidor, por lo que cuenta con peticiones (*requests*, por parte del cliente, y respuestas (*responses*), por parte del servidor. Para que un API se considere RESTful, es decir, que sigue los principios definidos en REST, tiene asegurar unas restricciones. Estas son: interfaz uniforme (*uniform interface*), cliente-servidor (*client-server*), sin estado (*stateless*), cacheado (*cacheability*) y sistema en capas (*layered system*). Las dos primeras ya se han explicado al hablar de recursos y el modelo cliente-servidor. La tercera implica que la interacción entre cliente y servidor se produce petición a petición, no se almacena ninguna información de sesión (al menos por parte del servidor). Cada petición tiene la información suficiente para realizar la operación pertinente, y el servidor no conoce las anteriores. Por último, *cacheability* y *layered system* hacen referencia a cacheo de respuestas y a la distribución transparente para el cliente de los distintos componentes del API en distintos servidores, respectivamente. Ambas no son pertinentes en este proyecto.

Otro concepto importante es la manera de interactuar entre cliente y servidor. REST sigue las cuatro operaciones básicas de almacenamiento conocidas por sus siglas

en inglés como CRUD (*create*, *read*, *update* y *delete*). En HTTP se implementan con los métodos *POST* para *create*, *GET* para *read*, *PUT* o *PATCH* para *update* y *DELETE* para *delete*.

En este proyecto, dados los requisitos, los usos de las operaciones son los siguientes:

- GET: se obtiene información del recurso especificado. Mediante parámetros se puede refinar la información que se obtiene. Por ejemplo, en el caso de la lista de artistas, como se quiere obtener artistas por initial, en el parámetro `initial` se especifica la letra.

- POST: creación de contenido. En el caso del proyecto, el usuario puede crear comentarios, añadir vídeos como favoritos, añadir artistas como favoritos y registrarse. Los parámetros van en el cuerpo de la petición, a diferencia del método GET, donde van en la misma URL. Además se utiliza para el login, para obtener un token.

- PUT: se utiliza para actualizar información, en este caso la única información que se actualiza es la del perfil del usuario.

- DELETE: como el nombre indica, para borrar contenido. En los requisitos funcionales no se ha contemplado el borrado de contenido, por lo que no se utiliza. En versiones futuras de la aplicación es posible que se incluya el borrado de comentarios. En ese caso se utilizaría este método.

### 4.1.1. Conjunto de recursos

Como se ha visto, es esencial identificar los recursos necesarios. En este caso, a partir de los requisitos se han obtenido los siguientes, mostrado en la tabla 4.1. Están definidos con su URI asociada.

El conjunto de operaciones asociado a cada recurso se encuentra en el Anexo B.

### 4.1.2. Modelos de datos

Una vez se han identificado los recursos del API, se pueden diseñar los modelos que maneja el servidor. Se ha decidido utilizar 2 conjuntos de modelos de datos, el primero corresponde a los modelos que a través de *Entity Framework* se utilizan para interactuar con la base de datos, y el segundo, que está formado por objetos de transferencia de datos (*DTO*, en inglés), que se utilizan en las peticiones y respuestas al API.

Los siguientes corresponden a los objetos de *Entity Framework*. Artista: Almacena la información de los artistas de la plataforma. Los campos que contiene son:

| ***URI*** | ***Descripción del recurso*** |
|---|---|
| `/api/sessions` | Sesiones. Se trata de los vídeos o audios que se reproducen en la aplicación. |
| `/api/artists` | Artistas/pinchadiscos. Los autores de las sesiones, que contienen información sobre su trayectoria o imágenes. |
| `/api/users` | Usuarios de la plataforma *Pocketbeat*. |
| `/api/search` | Recurso utilizado para representar la búsqueda de tanto sesiones como vídeos. Aunque no es recomendado utilizar acciones como nombres de recursos, en este caso es necesario, puesto que no se puede asociar la acción de búsqueda a sólo las sesiones o los artistas, ya que se requieren ambos en los resultados. |
| `/api/auth` | Se utiliza para la gestión de la autenticación de los usuarios. |

Tabla 4.1: URIs y recursos

- Identificador: entero.
- Usuario: los artistas también son usuarios de la plataforma y este campo establece la relación.
- Descripción: cadena de texto.
- Destacado: entero.
- Sesiones: lista de sesiones de las que es autor.
- Géneros: lista de los géneros con los que se asocia su música.
- Redes sociales: apunta a otra tabla con enlaces a las distintas redes sociales que tiene el artista.
- Seguidores: lista de usuarios seguidores de este artista.

Usuario: Almacena la información de los usuarios de la plataforma. Los campos que contiene son:

- Identificador: entero.
- Nombre: cadena de texto.
- Apellidos: cadena de texto.
- Nombre de usuario: cadena de texto.
- Correo electrónico: cadena de texto.
- Género: masculino, femenino u otro. Entero.
- País: cadena de texto.
- Imagen: referencia a la imagen de avatar del usuario.
- Estatus: indica si el usuario está activo, sin activar, suspendido. Enumeración.
- Hash contraseña: hash de la contraseña del usuario, utilizando SHA-512.
- Artista: Si se trata de un usuario que también es artista, tiene una referencia.
- Artistas favoritos: lista de artistas seguidos.

Imagen: Almacena la información de imagen de un usuario o un artista. Los campos son:

- Identificador: entero.
- Nombre fichero: nombre del fichero almacenado en el servidor. A diferencia de las sesiones, las imágenes se almacenan en el servidor Windows.

Sesión: Almacena la información que corresponde a una sesión de un artista. La información que almacena es:

- Identificador: entero.
- Artista: referencia al autor de la sesión.
- Nombre: cadena de texto.
- Nombre URL: URL que identifica la sesión desde el navegador. Cadena de texto.
- Estado: las sesiones se pueden subir y estar sin aprobar, y por tanto no accesibles por los usuarios. Enumeración.
- Fecha de creación: fecha.
- Imagen portada: referencia a la imagen que se muestra de forma estática antes de reproducir la sesión.
- Duración: segundos. Entero.
- Descripción: cadena de texto.
- Destacado: entero.
- Nombre fichero: cadena de texto.
- Extensión: cadena de texto.
- Altura fichero: número de píxeles de altura del vídeo. Entero.
- Anchura fichero: entero.
- Tamaño fichero: bytes, número de coma flotante.
- Géneros: lista de referencias a géneros musicales.
- Escuchas: lista de referencias a Escucha, explicado más adelante.

Comentario: Almacena la información de un comentario realizado por un usuario en una sesión. Contiene:

- Identificador: entero.
- Sesión: referencia a Sesión en la que aparece el comentario.
- Usuario: referencia al autor.
- Estado: enumeración, activo o borrado.
- Texto: cadena de texto.
- Fecha creación: fecha.
- Fecha última modificación: fecha.

Género: Géneros musicales. Campos:

- Identificador: entero.
- Nombre: cadena de texto.
- Sesiones: lista de referencias a Sesión, sesiones del género musical.

Escucha: Información sobre escuchas de una sesión. Los campos que contiene son:

- Identificador: entero.
- Sesión: referencia a la Sesión.
- Dirección usuario: objeto con información sobre una dirección IP e información del usuario en caso de que esté registrado.
- Tiempo escucha: segundos, entero.
- Fecha escucha: fecha.

En la figura 4.1 se observa un diagrama UML con las relaciones y campos de las distintas clases.

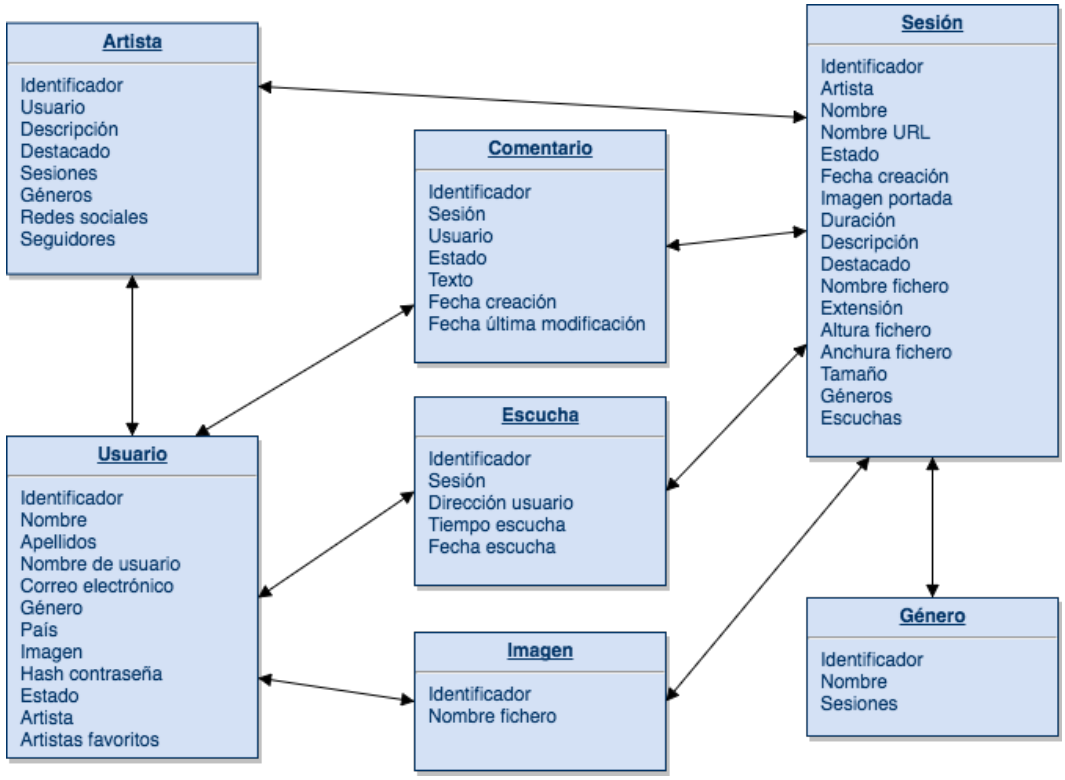


Figura 4.1: Diagrama UML de las clases que forman los modelos necesarios en la base de datos.

Tras detallar el conjunto de modelos que se han utilizado para la base de datos, se pasa a especificar algunos de los objetos de transferencia de datos que se han utilizado para las respuestas del API. Contienen únicamente la información necesaria para mostrar en la pantalla correspondiente de la aplicación móvil.

En concreto se van a especificar las respuestas de Artista y Sesión, en la tabla 4.2.

| ***Nombre del DTO*** | ***Campos*** |
|---|---|
| Artista | Contiene un número bastante inferior de campos al del modelo que se encuentra en la base de datos. Estos son:<br>– Identificador: entero que identifica al artista. Corresponde al mismo identificador que en la base de datos.<br>– Nombre: cadena de texto.<br>– Biografía: cadena de texto.<br>– Imagen: cadena de texto con una URL que corresponde al recurso de la imagen del artista.<br>– Sesiones: lista de DTOs de tipo Sesión que se especifica a continuación. |
| Sesión | De la misma forma que con Artista, es reducido en comparación a la información almacenada en la base de datos. Los campos son:<br>– Identificador: entero.<br>– Título: cadena de texto.<br>– Descripción: cadena de texto.<br>– URL visionado: cadena de texto.<br>– Id Artista: entero.<br>– Nombre Artista: cadena de texto. |

Tabla 4.2: Modelos de objetos de transferencia de datos.

De la misma forma, en las peticiones al API se envían parámetros que se pueden encapsular en objetos. Estos objetos contienen campos que son asignados automáticamente, bien desde el cuerpo de la petición o la misma URI (dependiendo del método HTTP utilizado), cuando un controlador4.1.3 recibe una petición, a través del deserializador JSON que se ha nombrado anteriormente.

### 4.1.3. Separación en controladores

A partir del diseño de los recursos en URIs se procede a explicar cómo se traslada esto al tipo de clase que se utiliza en ASP.NET Web API para gestionar las peticiones y enviar respuestas. Este tipo de clase se conoce como controlador, *controller* en inglés. Se han creado 4 controladores: `SessionsController`, `ArtistsController`, `UsersController`, `AuthController`. Estas clases extienden la clase abstracta `ApiController`, que contiene un conjunto de métodos que son muy útiles, especialmente para las respuestas. Un ejemplo es el método `Ok()`, que envía una

respuesta con código HTTP 200, y se puede parametrizar para incluir en el cuerpo la información que se desee. Otro ejemplo es el método `Created()`, con el código 201, que se puede adaptar de la misma manera.

Otra de las funcionalidades del *framework* que resulta útil es la posibilidad de utilizar atributos (en la documentación en inglés se conocen como *attributes*) para declarar las clases y métodos HTTP que acepta cada método y la ruta o URI. Se puede declarar un prefijo para todos los métodos dentro de una clase, y en cada método un sufijo. En este ejemplo se muestra para el controlador de los artistas.

```
[RoutePrefix("api/artists")]
public class RestArtistController : ApiController
{
    private readonly IPocketBeatContext context;
    public RestArtistController(IPocketBeatContext context)
    {
        this.context = context;
    }
    [HttpGet]
    [Route("")]
    public GetArtistsResponse GetList(String ?initial)
    {
        // devuelve lista de artistas segun inicial, si se proporciona
    }
    [HttpGet]
    [Route("{id}")]
    public GetArtistResponse GetById(int id)
    {
        // devuelve artista que tiene dicho id
    }
}
```

Como se puede observar, para el método `GetList()` la URI es `/api/artists/` y para el método `GetById()`, si se consultara con el identificador 100, por ejemplo, es `/api/artists/100`. También se pueden utilizar atributos para forzar el uso de HTTPS y para autenticar a usuarios.

### 4.1.4. Seguridad

Uno de los requisitos no funcionales de la aplicación es el cifrado de la comunicación entre el cliente y el API Web. Para ello se ha configurado la aplicación para utilizar SSL y únicamente aceptar peticiones realizadas mediante HTTPS. La autenticación con el cliente se realiza mediante el uso de tokens. Mediante la solicitud a `/api/auth/login` se genera un token que el cliente recibe como respuesta y deberá incluir en las peticiones que requieren autenticación, como crear un comentario. Este token se incluye en la cabecera de la solicitud al API, en concreto en la cabecera `Authorization`, con el valor `Bearer <Token>`.

#### Autenticación mediante Tokens

Se ha seleccionado el estándar JSON Web Token[29] (JWT a partir de aquí) para la creación de tokens de acceso de los usuarios. Se ha utilizado porque se adecúa a las restricciones del estándar REST, ya que no se mantiene ningún tipo de estado, se envía con la petición y contiene la información necesaria en sí mismo. No es necesario almacenarlo de forma persistente en el servidor, lo que conlleva no tener que consultar la base de datos con cada petición, como se haría si se utilizara Autenticación Básica (*Basic Authentication*[1], en inglés). Esto reduce el tiempo de respuesta y la carga de la base de datos.

Los JWT tienen una estructura dividida en 3 partes: cabecera, carga (o *payload*) y firma. En la cabecera se indica el algoritmo de firma utilizado, como HMAC-SHA256. En el *payload* se incluye la información que se quiere comunicar con el token, como el nombre de usuario, su rol o la fecha de expiración. Mediante el uso de una clave secreta que el servidor almacena, se cifra la parte del token correspondiente a la cabecera y *payload*, que van codificados en Base64 y concatenados por un punto. El token resultante es la cabecera, el *payload* y la firma, todos codificados mediante Base64 y separados por puntos.

Para el uso de este proyecto, se ha incluido la información (en la terminología de JWT a cada campo en el *payload* se le conoce como *claim*) de identificador de usuario y fecha de expiración. En la implementación se ha utilizado el paquete NuGet System.IdentityModel.Tokens.Jwt[30] para la creación, serialización y validación de los tokens.

En el diagrama de la figura aparece la interacción entre cliente y servidor respecto a los JWTs. En el caso de este API la URI es `/api/auth/login`.

[1]Método de autenticación donde se envían los credenciales (por ejemplo usuario y contraseña) en una petición HTTP, mediante el uso de la cabecera `Authorization`, seguido de `Basic` y a continuación el nombre de usuario y contraseña concatenados separados por 2 puntos y codificados en Base64.

### 4.1.5. Gestión de imágenes

Como se ha explicado anteriormente, las imágenes de las sesiones, usuarios y artistas se comparten con la aplicación móvil asignándoles una URL. El almacenamiento de estas se encuentra en el servidor Windows que corre la aplicación Web y el API. La aplicación Web cuenta con un Web Handler que procesa las peticiones referentes a imágenes. En la petición se puede especificar las dimensiones de la imagen final. Esto permite cortar las imágenes o cambiar sus dimensiones al tamaño deseado, usando la librería ImageResizer. Por tanto el API envía en una cadena de caracteres la URL asociada a la imagen, consultada en la base de datos, y el Web Handler provee este recurso cuando se realiza una petición desde la aplicación móvil.

Por otra parte, las imágenes de avatar que utilizan los usuarios se envían al servidor codificadas en Base64, por su sencillez. Esta codificación tiene el inconveniente de que incrementa el tamaño de la imagen. El tamaño final se puede aproximar a 4/3 del tamaño original. En este caso se trata de imágenes de avatar con unas dimensiones reducidas, por lo que no supone un problema de rendimiento.

## 4.2. Infraestructura de integración continua

La configuración de la infraestructura de integración continua es también una parte importante del desarrollo. Es una inversión de tiempo que por otra parte es esencial para la puesta en marcha del API, y que permite crear un entorno de trabajo en el que los cambios en el código sean desplegados con una gran frecuencia y rapidez. Como ya se ha comentado en la sección 3.4, las tecnologías utilizadas para la permitir integración continua son el repositorio de código remoto, utilizando Bitbucket, el servidor de compilación automática TeamCity y el gestor de despliegues Octopus.

Es importante señalar que en el repositorio remoto se utilizan 2 ramas para el código que ha de ser compilado automáticamente. La rama `master` se utiliza para la compilación de paquetes que van destinados a la aplicación remota de pruebas, y la rama `rtm`[2] que se utiliza para la producción. Más adelante, en el capítulo 5 se explica en el entorno de pruebas la existencia de la aplicación remota de pruebas, no únicamente en local.

Desde el repositorio Bitbucket se ha creado una llave SSH que utiliza el servidor TeamCity para autenticación. Se han configurado 2 proyectos en TeamCity, uno para pre producción y el de producción. Desde los ajustes del proyecto de pre producción se

[2] *Release to manufacturer*, versión final de un *software* que se entrega a un fabricante, para distribuirlo. En el caso de software que no es distribuido físicamente, simplemente representa la versión final.

selecciona el origen desde el cual se obtiene el código, es decir, la URL del repositorio remoto en Bitbucket, con la rama `master`. Se realiza lo mismo para el proyecto de producción, pero con la rama `rtm`. El servidor TeamCity comprueba si hay cambios nuevos cada 60 segundos. A continuación, se ha configurado el conjunto de pasos que se realiza junto a la compilación. El primer paso es la descarga de los paquetes NuGet que necesita la solución. El segundo es la construcción de paquetes NuGet del propio proyecto, que se utilizan para enviar al servidor Octopus.

Para publicar los paquetes NuGet al servidor Octopus basta especificar la URL para el *feed* de este servidor y una llave que se puede crear desde este servidor también. El método que se utiliza para la publicación se conoce como NuGet Push, está explicado aquí[32].

Gracias a las configuraciones de ASP.NET, se puede transformar la configuración del API mediante el uso de ficheros `Web.config`. Estas transformaciones de la configuración permiten, entre otras, cambiar la base de datos que utiliza la aplicación. Desde TeamCity se puede qué configuración debe de usar NuGet para construir los paquetes. En el caso de pre producción se utiliza la configuración Release.Test, que se conecta a una base de datos remota de pruebas, y en el de producción la configuración Release.Live.

En el servidor Octopus se cuenta por tanto también con dos proyectos, que corresponden a los proyectos de TeamCity. Octopus obtiene los paquetes de su propio repositorio, ya que se han publicado allí desde TeamCity. Desde de la configuración de cada proyecto se han añadido los pasos necesarios para desplegar las aplicaciones. El primer paso es inicializar una variable con el valor de la fecha y hora actual, para crear una carpeta con ese nombre que contendrá la aplicación desplegada. El segundo paso es el despliegue, que consiste en un script utilizando PowerShell, que provoca que desde IIS la aplicación apunte al directorio virtual que se ha creado, con la fecha y hora anteriores como nombre. La máquina que ejecuta el servidor IIS que corre el API está configurado como un tentáculo (*tentacle*, en terminología de Octopus) que escucha en un puerto a peticiones del servidor Octopus, para recibir paquetes a desplegar.

Finalmente, para enseñar lo sencillo que es realizar un despliegue desde Octopus, se muestra en la siguiente figura 4.3. Después de pulsar el botón, queda incluir un número de versión, la posibilidad de incluir una reseña, como se ve en la imagen 4.4. A continuación se puede elegir un despliegue instantáneo o planificado, como muestra la imagen 4.5.

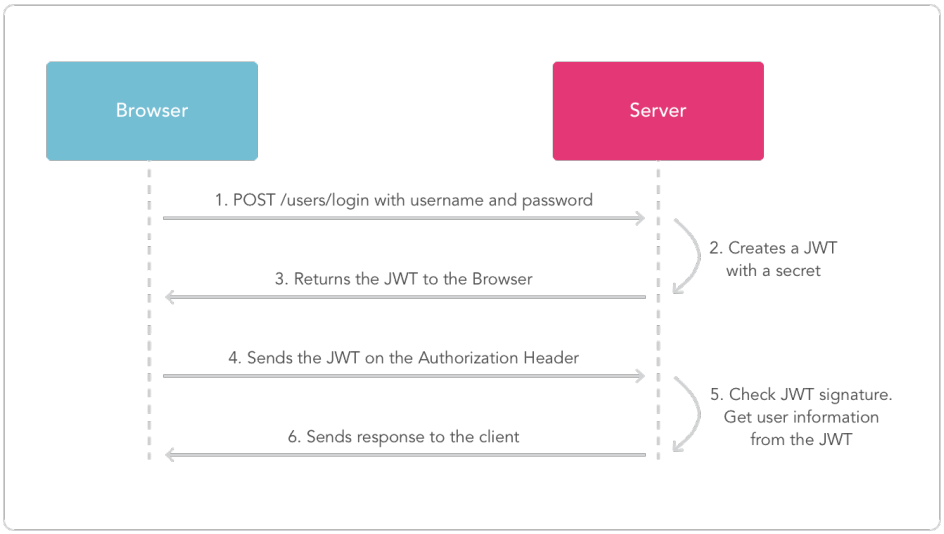


Figura 4.2: Interacción entre cliente y servidor utilizando *tokens*. Imagen obtenida de la página de JWT[31].

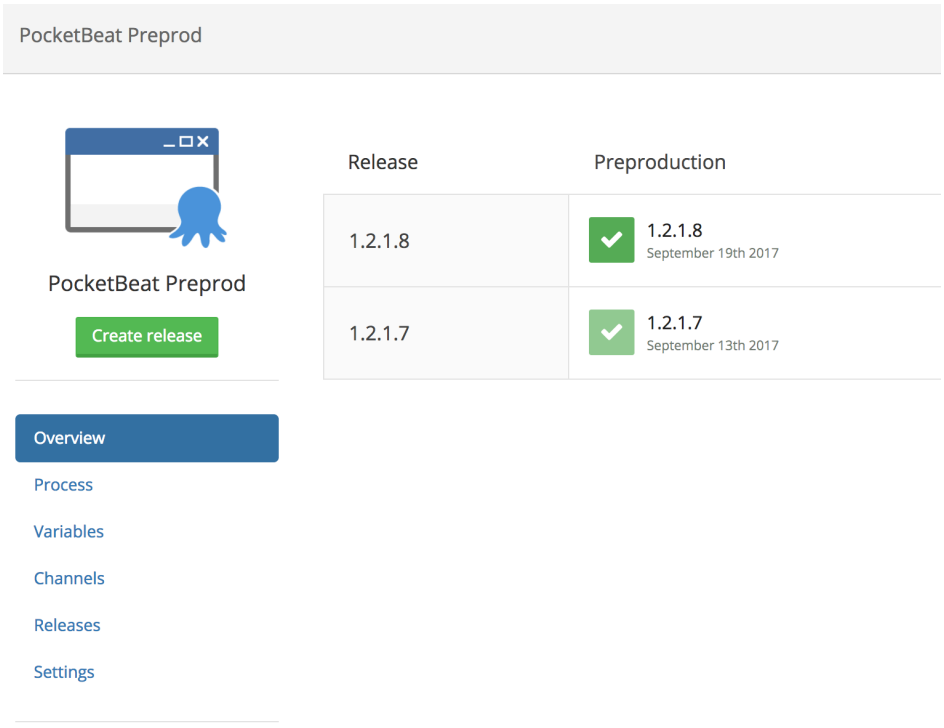


Figura 4.3: Creación de un despliegue desde Octopus Deploy.

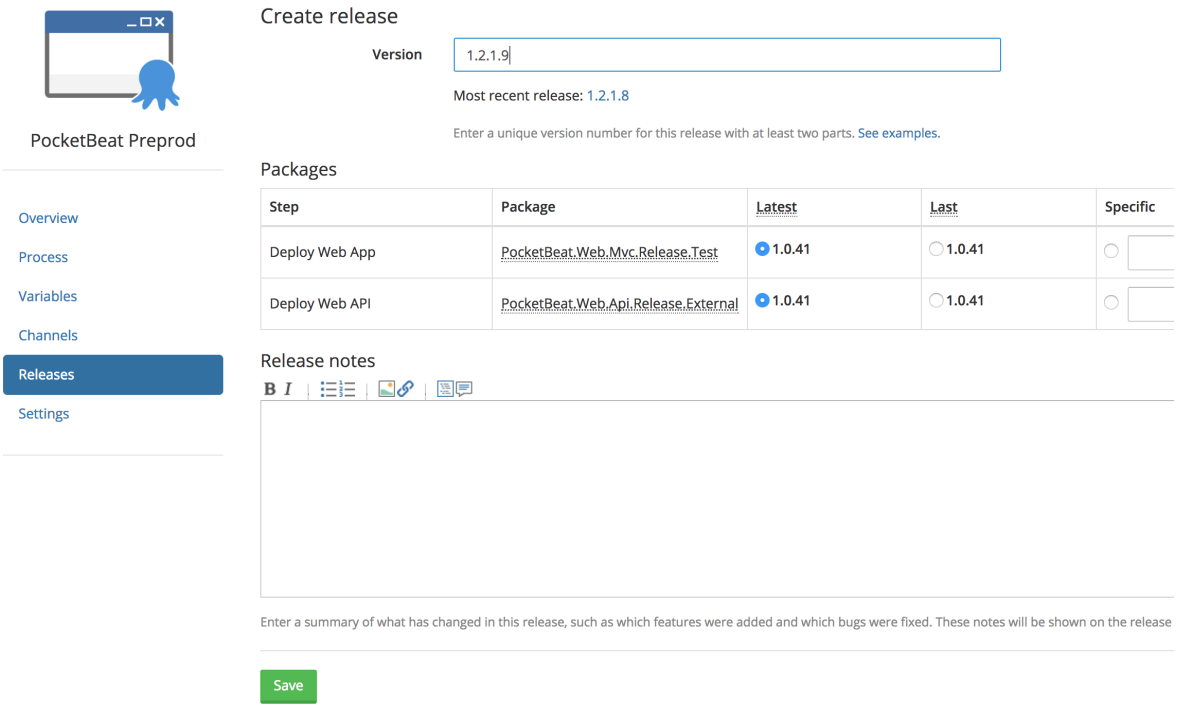


Figura 4.4: Creación de un despliegue desde Octopus Deploy.

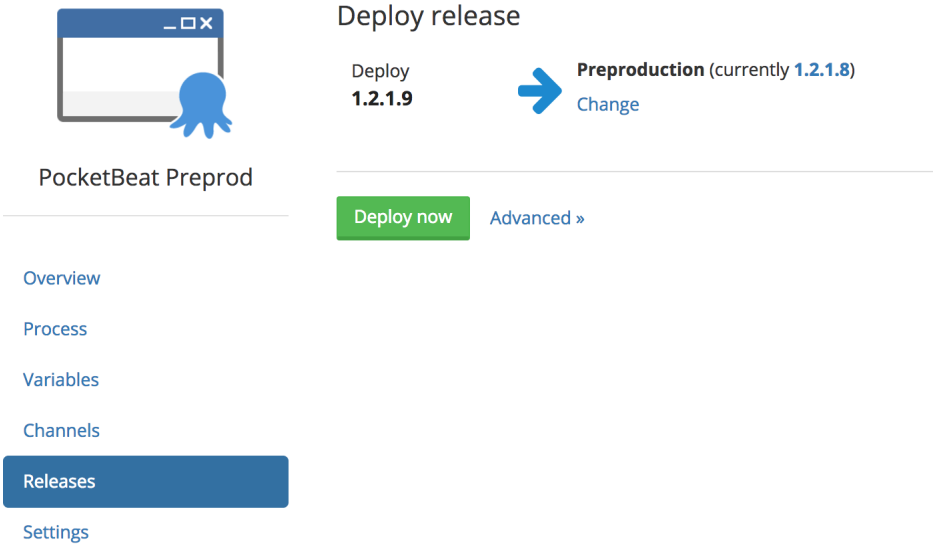


Figura 4.5: Creación de un despliegue desde Octopus Deploy.

### 4.2.1. Instancias en Amazon Web Services

El servidor de integración continua, que ejecuta tanto TeamCity como Octopus Deploy, es una máquina virtual Windows tipo t2.medium[33], tiene 2 procesadores virtuales y 4 GiB de memoria RAM. Cuenta con un disco duro SSD de 80 GB. El máquina del servidor IIS es del mismo tipo, con un disco duro de 60 GB.

# 5. *Validación*

En este capítulo se describe la fase de pruebas para comprobar que el funcionamiento del API desarrollado es el propuesto en los requisitos. Primero pruebas no formales realizadas mediante la herramienta Postman, que permite realizar llamadas a APIs (especificando la URI, el método HTTP, el conjunto de parámetros, el cuerpo, etc.) mientras se ejecuta el API en Visual Studio, y luego ya formalizadas, utilizando pruebas unitarias para cada método de los controladores. Las pruebas unitarias forman parte del *framework* .NET, no se trata de ningún paquete de terceros. Por último, también se explica el despliegue del API en el servidor, para probarla en un entorno idéntico al de producción.

## 5.1. Debug mediante Visual Studio y Postman

Mientras se ha realizado el desarrollo del API, se ha ido comprobando con cada controlador que los métodos realizaban lo que debían. Para esto es importante mencionar 2 herramientas que han resultado de gran utilidad.

La primera es el modo de depuración de Visual Studio. Mediante este entorno de programación se puede trazar las excepciones a su origen en el código, consultar el valor de las variables en cualquier momento, utilización de *breakpoints*, etc. Otra funcionalidad que ha resultado útil es la definición de distintos ficheros de configuración para la aplicación, que se han utilizado para diferenciar entre 3 situaciones. La primera es la versión local de la aplicación. Para esta se establece una conexión a una base de datos local utilizando Microsoft SQL Server Express. Además se cuenta con otros dos ficheros de configuración, uno para el servidor de pruebas, cuya conexión es a una base de datos remota, con un catálogo[1] de prueba, y por último, la versión de producción, con la conexión al catálogo de producción.

La herramienta Postman se ha utilizado para realizar solicitudes mientras la ejecución del API de forma local. Se trata de una herramienta específica para esto,

[1]La base de datos está dividida en catálogos. Cada catálogo tiene las mismas tablas creadas para el proyecto pero distinta información. La base de datos es la misma, alojada en AWS como se ha mencionado previamente.

mucho más cómoda que realizar peticiones utilizando un navegador Web o la línea de comandos con herramientas como curl. En este caso se ha utilizado para editar la cabecera de cada petición, los parámetros y el cuerpo, y leer las respuestas de una forma clara. Además cuenta con un historial de peticiones, para poder realizar peticiones en pruebas anteriores sin la necesidad de introducir la información de nuevo.

## 5.2. Conjunto de pruebas unitarias

Las pruebas se han formalizado creando un proyecto de pruebas unitarias en Visual Studio. Se han creado métodos de prueba para cada controlador. Mediante el uso de la clase `Assert` de la librería de pruebas se utilizan los métodos `IsNotNull`, `IsInstanceOfType` y `AreEqual` para comprobar que el resultado de cada método de los controladores no es nulo o indefinido, es del tipo y contiene la información esperada. Se utiliza información de prueba, no afecta a la base de datos en producción.

Para ilustrar un método de prueba se puede observar en este ejemplo, donde se comprueba que obtener una sesión por identificador produce como resultado esa sesión.

```
[TestMethod]
public void GetSession_ShouldReturnCorrectSession()
{
    var testSessions = GetTestSessions();
    var controller = new SessionsController(testSessions)
    var result = controller.GetById(10) as
        OkNegotiatedContentResult<Session>;
    Assert.IsNotNull(result);
    Assert.AreEqual(testSessions[10].Id, result.Content.Id);
}
```

El conjunto de pruebas que se ha construido es:

- `SessionsController`: el listado de las sesiones contiene todos los resultados, en el orden especificado. La sesión identificada por id contiene la información que debe. Obteniendo todos los comentarios de prueba de una sesión se obtienen estos, la creación de un comentario es correcta y se devuelve un código HTTP 201. No se puede crear un comentario con texto vacío o de un usuario no existente o no autentificado.

- `ArtistsController`: tanto la colección de artistas recuperada como artistas individuales y sus sesiones corresponden a las de prueba.

- `UsersController`: se obtiene la información de un usuario de prueba y el registro es correcto y se devuelve el código HTTP 201. No se puede registrar un usuario con un correo existente en el sistema o con información no válida en los campos de registro (correo no válido, nombre vacío, etcétera). La modificación de la información utilizando PUT devuelve el código 200 (Ok). En caso de que el usuario no sea correcto o no esté autentificado el código es de error.

- `SearchController`: la búsqueda de sesiones y/o artistas produce el resultado esperado con los datos de pruebas.

- `AuthController`: la creación de un token para un usuario con una cuenta válida (correo existente y contraseña correcta) es satisfactoria, y se comprueba que el token se puede utilizar para realizar una petición para la que se necesita autenticación, como la creación de un comentario.

En total se han programado 15 pruebas, que comprueban que el API funciona correctamente.

## 5.3. Despliegue de aplicación en servidor de prueba

La aplicación se ha desplegado primero de prueba, utilizando la infraestructura de despliegue de integración continua ya comentada. Se ha registrado un subdominio DNS para acceder a ella utilizando el servicio Route 53 de AWS. Desde el servidor IIS está configurada como una aplicación más, no se trata de un servidor distinto al de producción, pero hay una separación entre las distintas aplicaciones. Se han realizado solicitudes utilizando la herramienta Postman, que se había utilizado de forma local, y los resultados son los mismos. Se puede observar un sencillo esquema sobre esto en la figura 5.1.

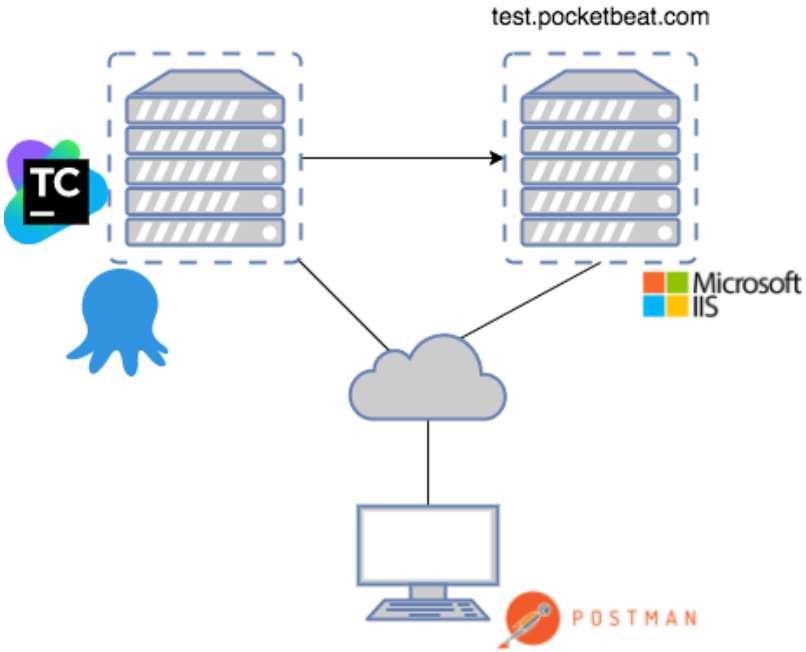


Figura 5.1: Diagrama del despliegue en el servidor de prueba.

# 6. *Conclusiones*

El objetivo de este capítulo es la explicación de la gestión del proyecto, las conclusiones del mismo, el trabajo futuro que podría realizarse y finalmente, la opinión personal por parte del estudiante que lo ha realizado.

## 6.1. Gestión del proyecto

El proyecto está dividido en las fases: familiarización con la arquitectura de la plataforma de Pocketbeat y sus tecnologías, análisis y diseño, implementación y pruebas, creación y uso de la infraestructura de despliegue y escritura de la memoria.

En la primera fase, se tuvo que realizar un estudio profundo de la arquitectura de la plataforma en uso, puesto que tiene una complejidad a la que el autor del proyecto no estaba acostumbrado previamente. Esto incluye conocimiento sobre el proveedor de servicios en la nube, Amazon Web Services. A su vez, la tecnología era nueva, no tanto por la sintáxis de C#, que guarda muchas similitudes con otros lenguajes orientados a objetos como puede ser Java, sino por tratarse de un *framework* Web del que no se tenían conocimientos previamente.

En la segunda fase, se realizó el análisis de requisitos, partiendo de los requisitos de la aplicación móvil para la cual el API está enfocada. También se decidió la arquitectura y tecnologías utilizadas de manera que la integración con las aplicaciones existentes fuera lo más sencilla posible. Se estudió las restricciones que se han de llevar a cabo si se quiere adoptar la arquitectura REST para un API Web y conforme a ellas se diseñó el conjunto de recursos y sus correspondientes URIs. Además se llevó a cabo un estudio de las distintas opciones de autenticación.

La tercera fase consistió en la implementación del diseño propuesto en la fase anterior, y el diseño y creación de pruebas formales para la validación de la aplicación. Para la creación de pruebas fue necesaria la consulta de documentación.

En la cuarta fase se llevó a cabo la creación de la infraestructura de integración continua o de despliegue. Se desplegó la versión más avanzada hasta la fecha del API, que incluía todos los requisitos iniciales. Primero se realizó en una aplicación remota de pruebas y más tarde en producción.

Por último, se terminó la redacción de la memoria

En la tabla 6.1 se puede observar el reparto de horas que corresponde a cada fase del proyecto.

| ***Fase*** | ***Horas*** | ***Porcentaje*** |
|---|---|---|
| Familiarización plataforma y tecnología | 25 | 7.25 % |
| Análisis y diseño | 53 | 15.36 % |
| Implementación y pruebas | 175 | 50.72 % |
| Creación infraestructura integración continua | 34 | 9.86 % |
| Redacción de memoria | 58 | 16.81 % |
| **Horas totales** | 345 | 100 % |

Tabla 6.1: Esfuerzos

## 6.2. Conclusiones

El objetivo de desarrollo del API con los requisitos especificados por la aplicación móvil se ha conseguido con éxito, así como la infraestructura para el despliegue de futuras versiones. El despliegue tanto en entornos de pruebas como de producción se ha realizado con éxito, sin embargo la aplicación móvil se encuentra en fase de validación con usuarios probadores beta, no accesible a todo el público.

Una vez la aplicación sea lanzada al mercado podrá dar un mejor soporte a los usuarios móviles de la plataforma y cubrirá las carencias que otras soluciones tienen.

El uso de una tecnología como ASP.NET, una de las más usadas para desarrollo *backend*, conlleva que el proyecto pueda ser continuado por otros desarrolladores. Con el lanzamiento de .NET Core por parte de Microsoft, un proyecto de código libre y multi plataforma, la tecnología está atrayendo una gran cantidad de desarrolladores. Según la encuesta anual del servicio Stack Overflow[34], .NET Core es el *framework* que ocupa el tercer lugar en satisfacción por sus desarrolladores, por detrás de React y Node.js (ambos de JavaScript) y también el tercer puesto en popularidad detrás de Node.js y AngularJS.

El resultado del proyecto es por tanto un API completo y funcional que cumple las características deseadas.

## 6.3. Trabajo futuro

El trabajo futuro se podría centrar principalmente en temas de seguridad. Una mejora sustancial sería la utilización del *framework* para .NET llamado Identity Server.

De esta forma se centralizaría la autenticación de los usuarios entre la aplicación Web y el API.

Otro de los aspectos que podría considerarse es el cacheo de las respuestas del API, mediante el uso de ETags. Los ETags son números que se comparten utilizando cabeceras de HTTP. Este número representa el estado de la información del recurso solicitado. Desde el servidor se comprueba el valor del ETag de la petición y si se trata de un valor distinto al actual, se informa de ello en la respuesta. Si se trata del mismo valor, se informa igualmente y el cliente puede utilizar la información que ya había solicitado. De esta forma las peticiones se reducen de tamaño. Una vez implementado el cacheo, resultaría interesante la distribución del API en servidores en distintas áreas del mundo para garantizar un buen servicio en cualquier parte, no únicamente en Europa central donde se encuentran las máquinas virtuales actualmente. El contenido multimedia sin embargo está configurado a través de la red de entrega de contenidos de Amazon, Cloudfront, para estar cacheado en distintos centros de datos del globo, por lo que los datos que más tamaño tienen en la comunicación con la aplicación móvil están cubiertos.

En la validación del API no se ha incluido ninguna prueba de carga o rendimiento, con muchas peticiones simultáneas por ejemplo. Esto sería interesante para detectar cuellos de botella y diseñar mejoras en la arquitectura.

Por último, según la aplicación móvil incluya nuevas funcionalidades, el API tendrá que soportarlas. Para ello sería interesante utilizar un versionado del API, de forma que se pudiera distinguir entre las distintas versiones de la aplicación para procesar las peticiones acordemente.

## 6.4. Opinión personal

La realización de este proyecto ha sido un desafío en distintos aspectos, desde el uso de nuevas tecnologías hasta la gestión del proyecto, pasando por aspectos de seguridad que son especialmente delicados al tratarse de información de usuarios reales la que está en juego. Se ha constatado la utilidad de los conocimientos adquiridos en una gran variedad de asignaturas vistas en la carrera, la aplicación de una mezcla de ellos para lograr un objetivo. Por otra parte se ha utilizado una gran cantidad de tecnologías de terceros, como distintos paquetes y librerías, adaptadas a la solución que interesaba, algo que quizá no se había visto tanto en la carrera y que ha resultado tremendamente enriquecedor. El aprendizaje de todas estas nuevas tecnologías resulta muy útil y probablemente será una gran ayuda para poder progresar laboralmente.

En definitiva la finalización del proyecto supone una gran satisfacción para el autor

por lo positivos que son todos los resultados.

# 7. *Bibliografía*


[1] Página Web de Sobaka Developments. http://sobakadevs.com/. Consultado: 10 de Noviembre de 2017.

[2] Página Web de Pocketbeat. https://www.pocketbeat.com/. Consultado: 10 de Noviembre de 2017.

[3] Página Web de Xamarin. https://www.xamarin.com/. Consultado: 12 de Noviembre de 2017.

[4] Página Web de TestFlight. https://developer.apple.com/testflight/. Consultado: 12 de Septiembre de 2017.

[5] Página Web de Boiler Room. https://boilerroom.tv. Consultado: 25 de Septiembre de 2017.

[6] Canal de YouTube de Boiler Room. https://www.youtube.com/user/brtvofficial. Consultado: 25 de Septiembre de 2017.

[7] Aplicación para iOS de Boiler Room. https://itunes.apple.com/us/app/boiler-room-broadcasting-the-underground/id769578063?mt=8. Consultado: 25 de Septiembre de 2017. Publicada la versión 3.0.9 el 25 de Septiembre de 2017.

[8] Aplicación para iOS de WAV. https://itunes.apple.com/us/app/wav-watch-the-music/id1133388943?mt=8. Consultado: 25 de Septiembre de 2017. Publicada la versión 1.3 el 8 de Septiembre de 2017.

[9] Página Web de BE-AT.TV. https://be-at.tv/. Consultado: 10 de Noviembre de 2017.

[10] Aplicación para iOS de BE-AT.TV. https://itunes.apple.com/us/app/be-at-tv/id820786942?mt=8. Consultado: 25 de Septiembre de 2017. Publicada la versión 1.4 el 20 de Julio de 2015.

[11] ASP.NET. https://www.asp.net/. Consultado: 11 de Septiembre de 2017.

[12] Amazon Web Services. https://aws.amazon.com/. Consultado: 12 de Septiembre de 2017.

[13] AWS RDS. https://aws.amazon.com/rds/. Consultado: 11 de Septiembre de 2017.

[14] AWS EC2. https://aws.amazon.com/ec2/?nc2=h_m1. Consultado: 12 de Noviembre de 2017.

[15] Microsoft IIS. https://www.iis.net/. Consultado: 15 de Septiembre de 2017.

[16] ASP.NET MVC. https://www.asp.net/mvc. Consultado: 13 de Noviembre de 2017.

[17] Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/sql-server-2016. Consultado: 12 de Noviembre de 2017.

[18] Galería del gestor de paquetes NuGet. https://www.nuget.org/. Consultado: 12 de Septiembre de 2017. Publicado: 12 de Septiembre de 2017.

[19] Entity Framework. https://docs.microsoft.com/en-us/ef/. Consultado: 13 de Noviembre de 2017.

[20] Newtonsoft JSON.NET. https://www.newtonsoft.com/json. Consultado: 13 de Noviembre de 2017.

[21] AutoMapper. http://automapper.org/. Consultado: 13 de Noviembre de 2017.

[22] Apache log4net. https://logging.apache.org/log4net/. Consultado: 13 de Noviembre de 2017.

[23] Wowza Streaming Engine. https://www.wowza.com/products/streaming-engine. Consultado: 13 de Noviembre de 2017.

[24] Git. https://git-scm.com/. Consultado: 13 de Noviembre de 2017.

[25] Atlassian Bitbucket. https://www.atlassian.com/software/bitbucket. Consultado: 13 de Noviembre de 2017.

[26] TeamCity. https://www.jetbrains.com/teamcity/. Consultado: 13 de Noviembre de 2017.

[27] Automating Deployments with TeamCity and Octopus Deploy. https://blog.jetbrains.com/teamcity/2015/11/automating-deployments-with-teamcity-and-octopus-deploy/. Consultado: 14 de Noviembre de 2017. Publicado: 18 de Noviembre de 2017.

[28] Visual Studio. https://www.visualstudio.com/. Consultado: 13 de Noviembre de 2017.

[29] JSON Web Token. https://tools.ietf.org/html/rfc7519. Consultado: 20 de Septiembre de 2017. Publicado: Junio de 2017.

[30] System.IdentityModel.Tokens.Jwt. https://www.nuget.org/packages/System.IdentityModel.Tokens.Jwt/. Consultado: 20 de Noviembre de 2017.

[31] Diagrama JWT. https://cdn.auth0.com/content/jwt/jwt-diagram.png. Consultado: 14 de Noviembre de 2017.

[32] NuGet Push. https://octopus.com/docs/packaging-applications/package-repositories/pushing-packages-to-the-built-in-repository#PushingpackagestotheBuilt-Inrepository-UsingNuGet.exepush. Consultado: 20 de Septiembre de 2017.

[33] Instancias T2 de Amazon Web Services. https://aws.amazon.com/ec2/instance-types/t2/. Consultado: 20 de Septiembre de 2017.

[34] Stack Overflow Developer Survey 2017. https://insights.stackoverflow.com/survey/2017#technology. Consultado: 10 de Octubre de 2017.

[35] Octopus Deploy. https://octopus.com/. Consultado: 13 de Noviembre de 2017.

[36] ETag - HTTP. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/ETag. Consultado: 10 de Noviembre de 2017.

# Anexos

# A. *Mapa de la aplicación móvil*

En este anexo se incluyen todas las pantallas de la aplicación móvil de Pocketbeat, incluyendo anotaciones sobre la interacción con el usuario. La barra superior en color negro que contiene el logo de Pocketbeat cumple una función de botón universal para tener accesos directos en cualquier pantalla. Al pulsar una única vez la aplicación cambia a la pantalla anterior a la actual. Al pulsar dos veces la aplicación cambia a la pantalla inicial (tras registro o inicio de sesión). Al mantener pulsado se despliega el menú. Tras explicar esta interacción no se va a incluir en todas las pantallas para evitar la repetición.

Además, durante la reproducción de una sesión, se puede cambiar de modo vídeo a modo audio haciendo el movimiento con un dedo de desplazar hacia la izquierda o derecha sobre la zona del reproductor.

Las interacciones están explicadas con flechas rojas y el nombre e identificador de la figura que corresponde a la pantalla.

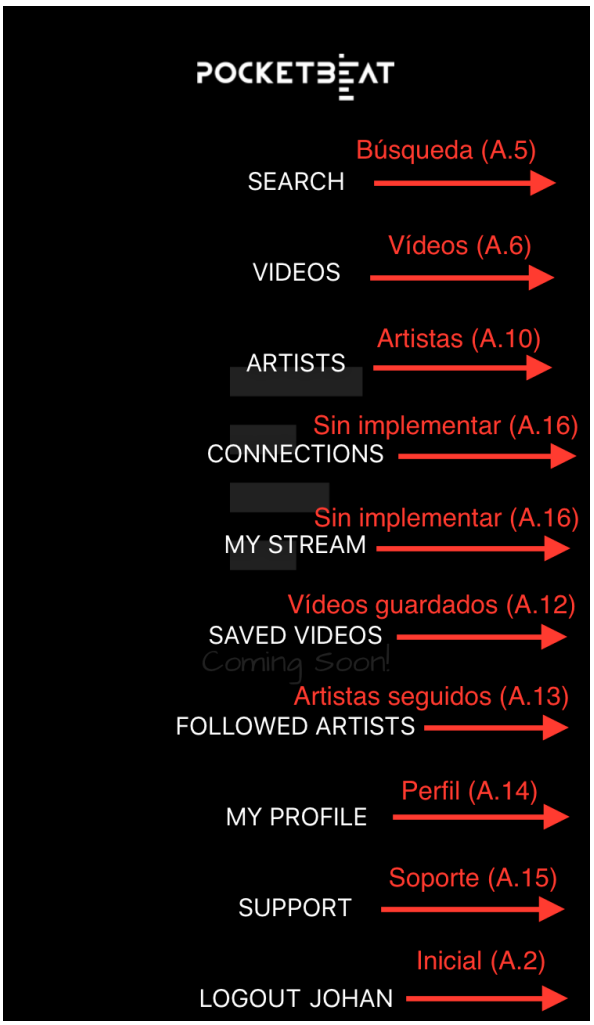


Figura A.1: Pantalla con el menú desplegable.

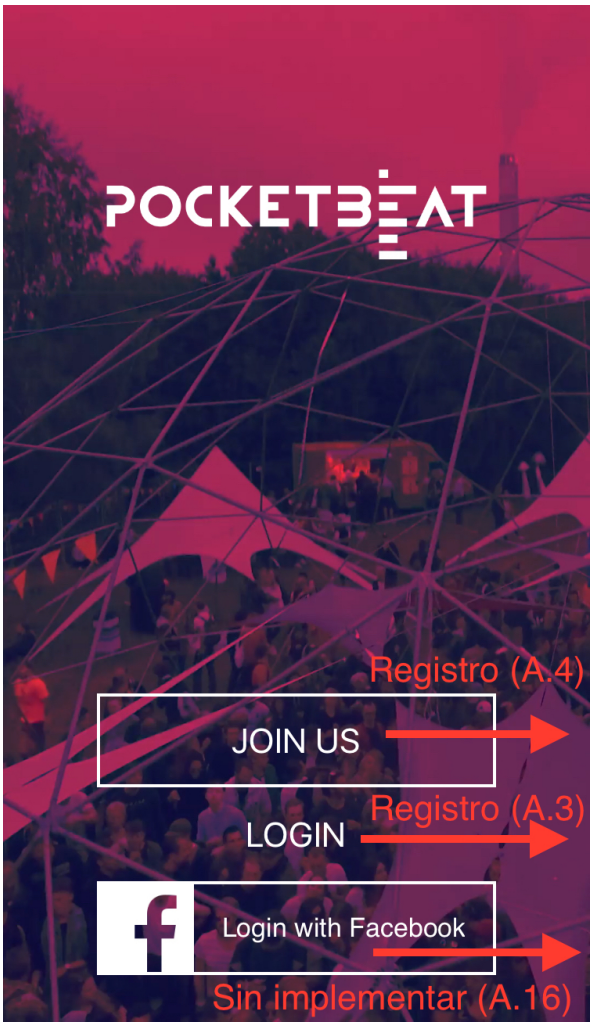


Figura A.2: Pantalla inicial tras abrir por primera vez la aplicación.

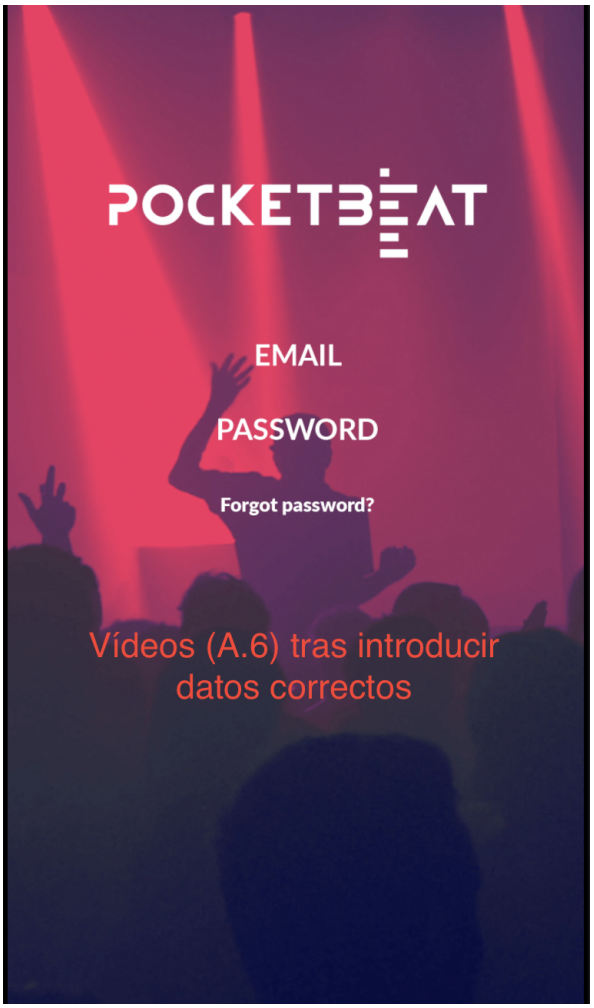


Figura A.3: Pantalla de inicio de sesión.

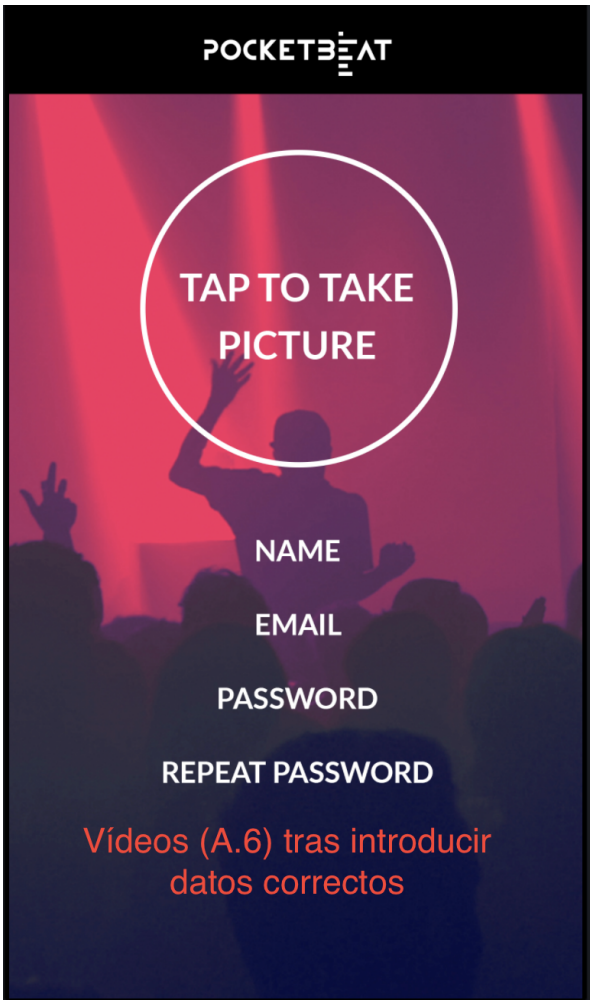


Figura A.4: Pantalla de registro.

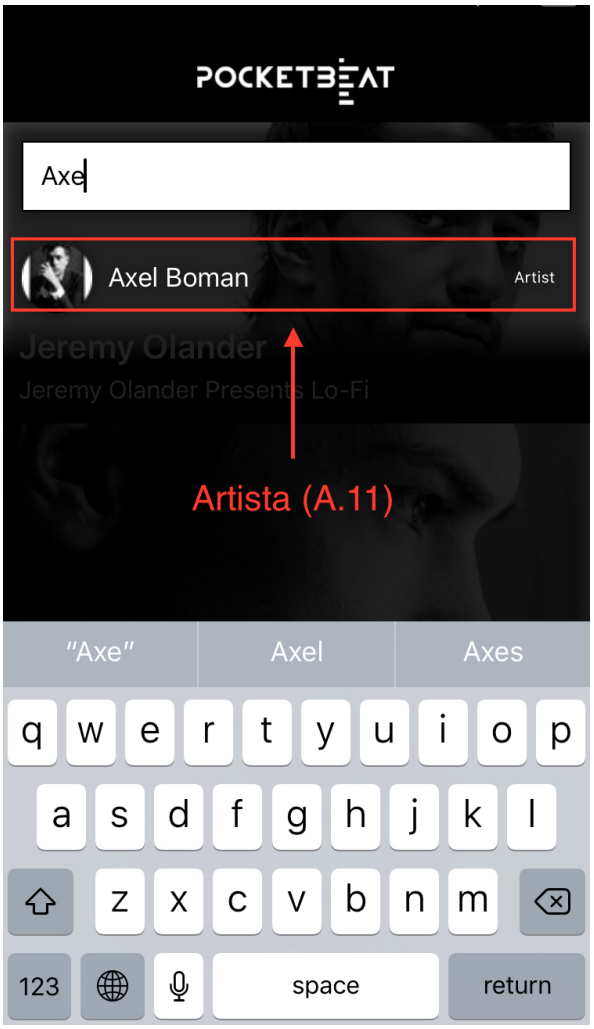


Figura A.5: Pantalla de búsqueda de artistas y sesiones.

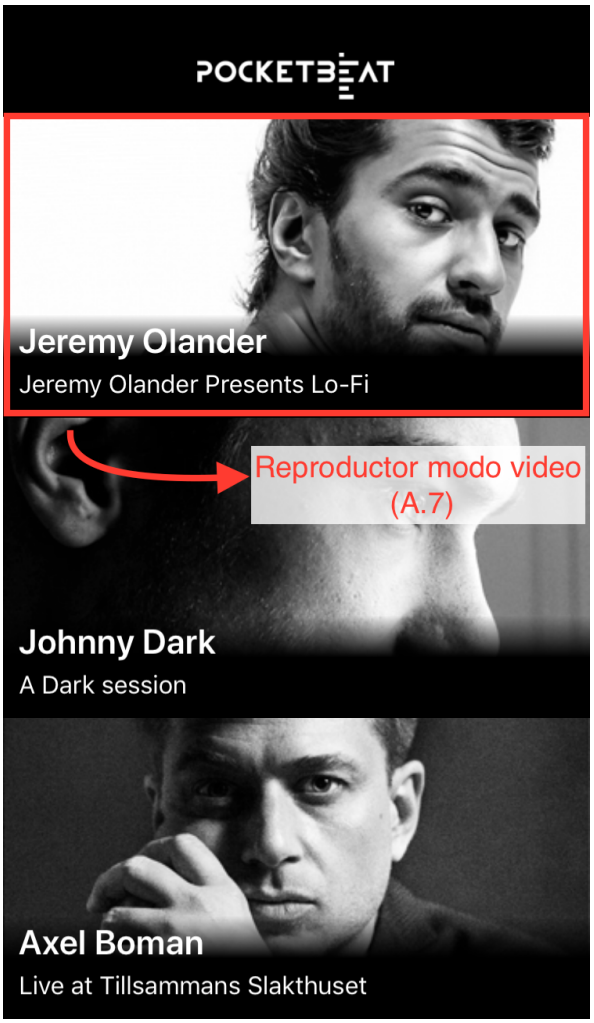


Figura A.6: Pantalla de los vídeos de la aplicación, una vez el usuario se ha identificado o registrado.

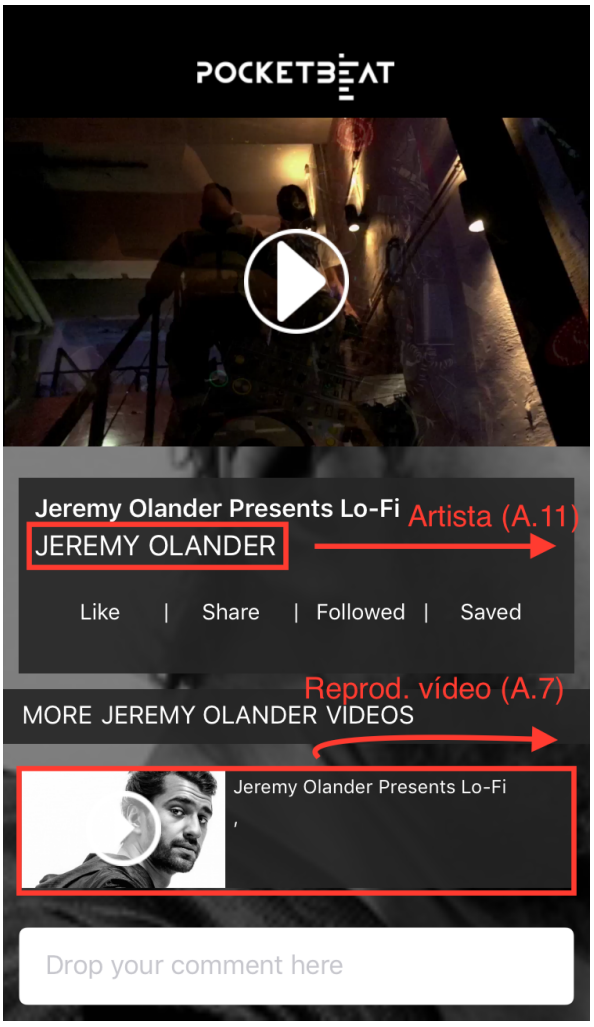


Figura A.7: Pantalla con el reproductor, en modo vídeo. Arrastrando el vídeo hacia abajo se minimiza en un reproductor pequeño, el de la figura A.9. De esta forma se puede seguir escuchando la sesión mientras se explora el contenido.

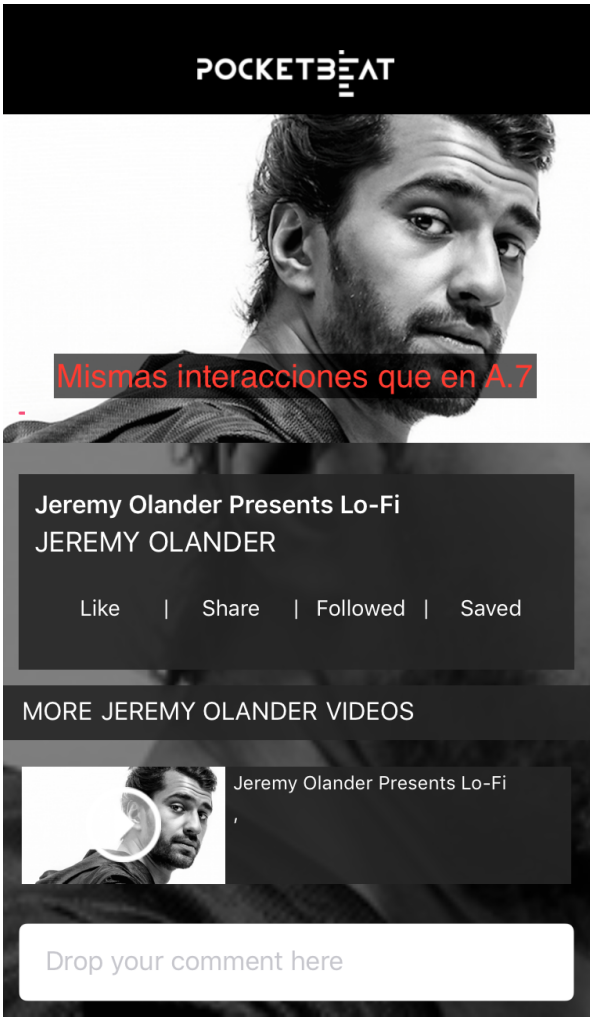


Figura A.8: Pantalla con el reproductor, en modo audio, donde se muestra una imagen estática. De la misma forma que en modo vídeo, se puede minimizar.


Figura A.9: Pantalla con el reproductor minimizado.

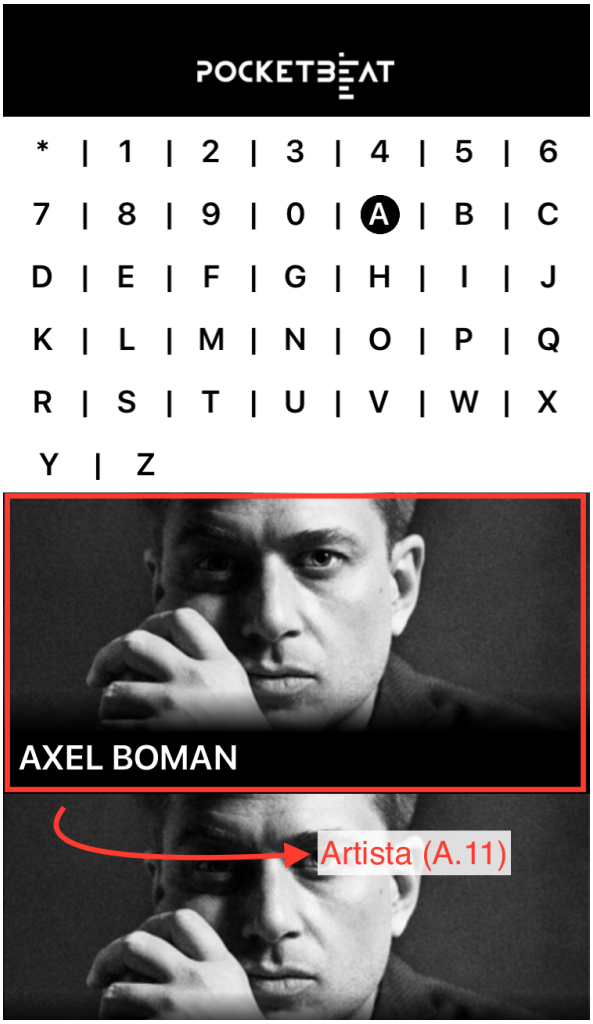


Figura A.10: Pantalla de la lista de artistas, se pueden filtrar por inicial seleccionando la letra en la zona superior con fondo blanco.


Figura A.11: Pantalla con la información de un artista.

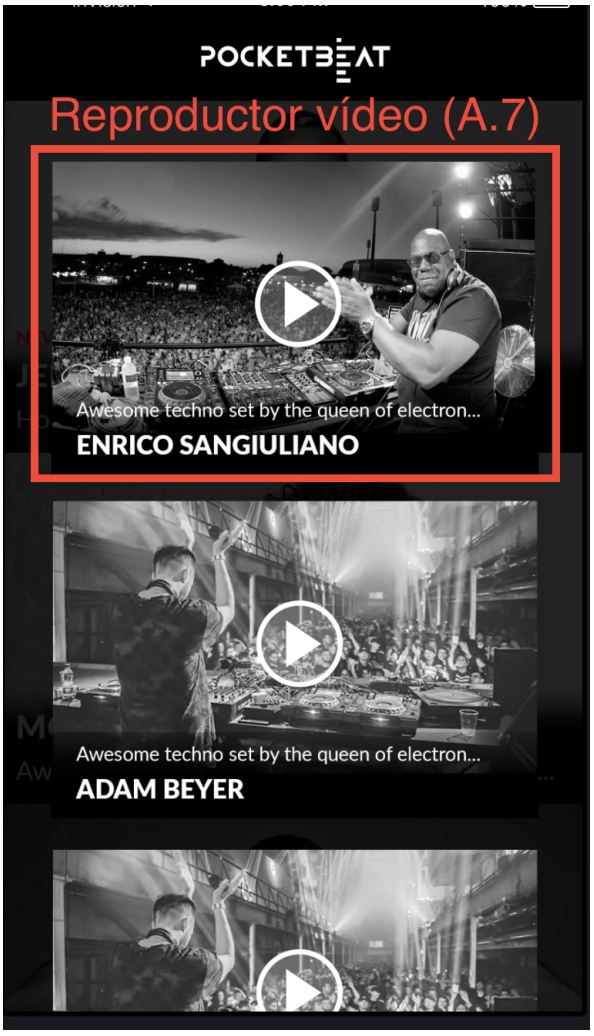


Figura A.12: Pantalla con los vídeos guardados por el usuario.

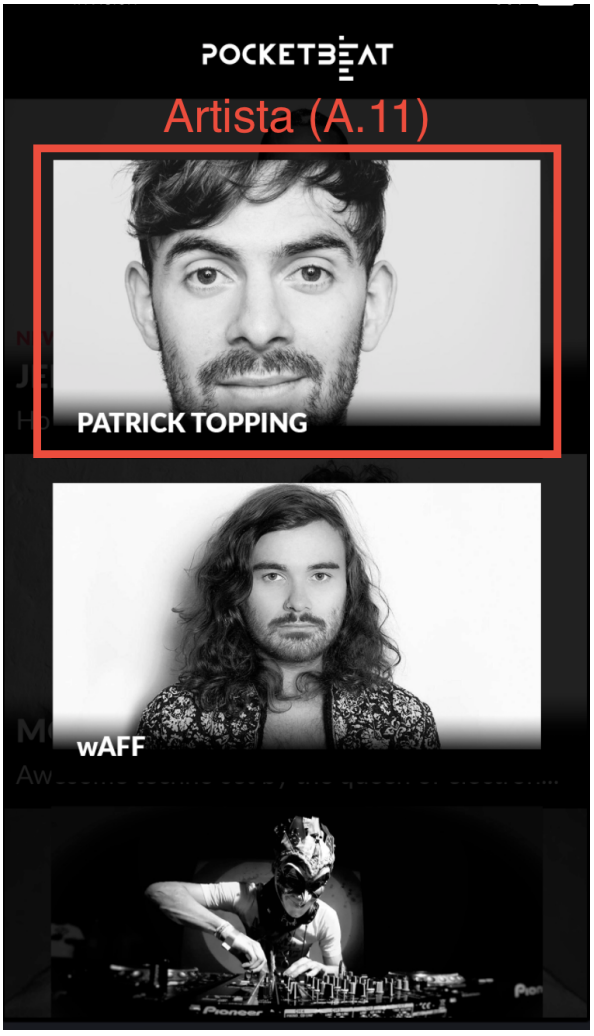


Figura A.13: Pantalla con los artistas seguidos.

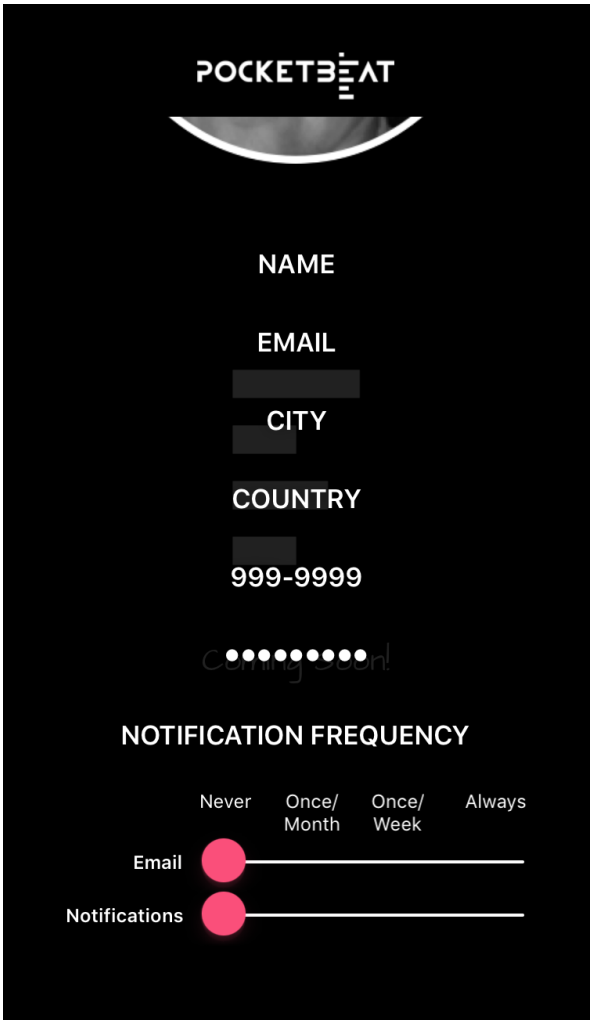


Figura A.14: Pantalla del perfil del usuario, donde se puede editar la información. No hay transiciones a otras pantallas; el guardado es automático.

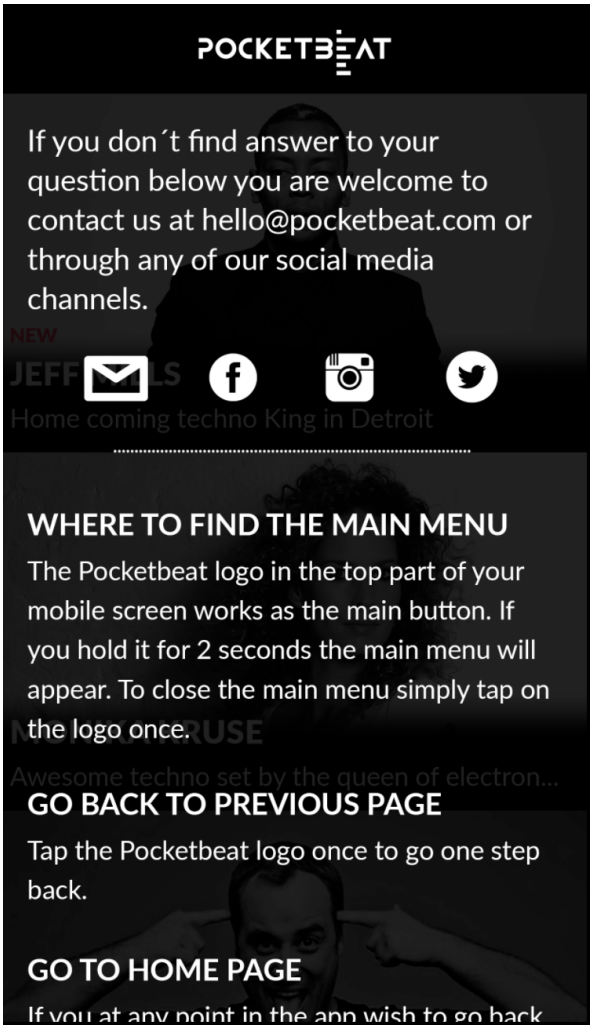


Figura A.15: Pantalla con información de soporte. Hay enlaces al correo electrónico y redes sociales de la plataforma.

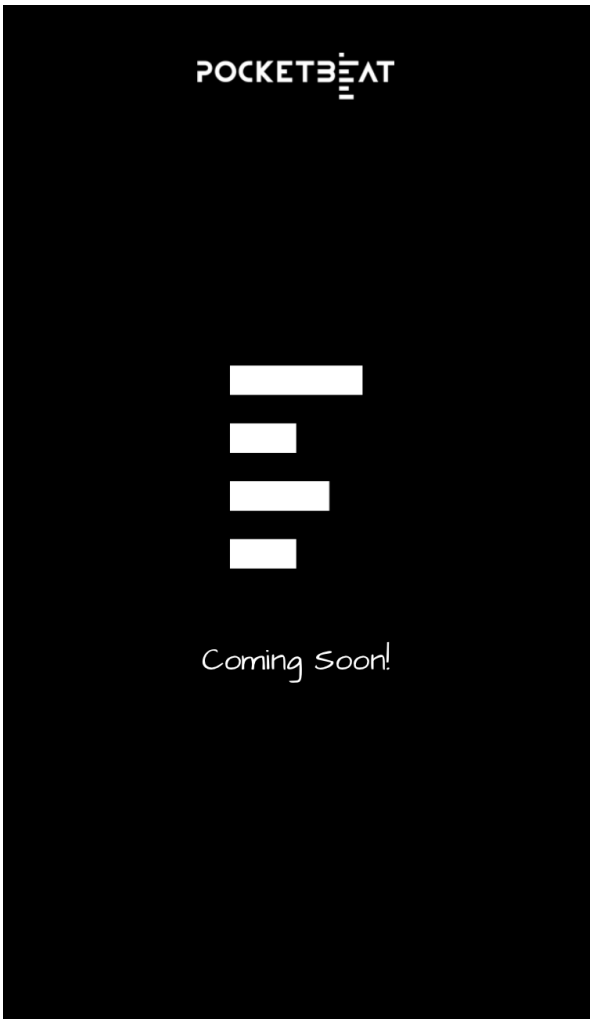


Figura A.16: Pantalla para funcionalidades sin implementar, como My Stream y Connections.

# B. *API RESTful de Pocketbeat*

En este anexo se especifican las operaciones admitidas por cada recurso, como se observa en la tabla B.1.

| ***URI*** | GET | POST | PUT | DELETE |
|---|---|---|---|---|
| /api/sessions | ✓ | | | |
| /api/sessions/{id} | ✓ | | | |
| /api/sessions/{id}/comments | ✓ | ✓ | | |
| /api/artists | ✓ | | | |
| /api/artists/{id} | ✓ | | | |
| /api/artists/{id}/sessions | ✓ | | | |
| /api/users/{id} | ✓ | | ✓ | |
| /api/users/ | | ✓ | | |
| /api/search | ✓ | | | |
| /api/auth/login | | ✓ | | |

Tabla B.1: Operaciones de REST admitidas por cada punto de acceso del API.

El primer punto de acceso, `/api/sessions`, mediante el método `GET` proporciona una colección de sesiones. Admite los parámetros *order, genre, count* y *from*. Los dos primeros se utilizan para ordenar las sesiones por los distintos métodos explicados en el análisis y filtrar por géneros musicales. Los dos últimos se utilizan para paginación.

El segundo, `/api/sessions/{id}`, responde con una sesión, cuyo identificador corresponde a *id*. Las URIs que corresponden a los artistas (`/api/artists` y `/api/artists/{id}`) responden de la misma manera, aunque la primera admite el parámetro *initial*, para el filtrado de artistas por su letra inicial. No admiten las operaciones POST o PUT porque la subida de sesiones y creación de artistas se realiza desde la aplicación Web.

Mediante `/api/sessions/{id}/comments` se accede a los comentarios de la sesión cuyo identificador corresponde a id. Usando GET se obtiene el conjunto de comentarios, usando POST se crea uno nuevo. Los parámetros para crear un nuevo comentario son userId y text (identificador de usuario y texto).

`/api/artists/{id}/sessions` se utiliza para obtener sesiones del mismo artista en la reproducción de una sesión, mediante el método GET.

Utilizando el punto de acceso `/api/users/{id}` se accede a la información de un usuario, identificado por id. Por el momento se utiliza para mostrar el propio perfil, por lo que la solicitud debe ir acompañada de un token que asegure proviene del usuario sobre el que se ha hecho la petición. Usando GET se obtiene la información, y utilizando PUT se modifica.

Con el método POST y la URI `/api/users/` se registran usuarios nuevos. En el cuerpo de la petición aparecen los campos que se utilizan en el modelo de usuario, que se han descrito previamente. Para subir la imagen se ha elegido codificarla en Base64 y mandar como parámetro, por su sencillez. Este método tiene el inconveniente de que incrementa el tamaño de la imagen. El tamaño final se puede aproximar a cuatro tercios del tamaño original. En este caso se trata de imágenes de avatar con unas dimensiones reducidas, por lo que no se trata de ficheros grandes que creen peticiones muy lentas por la red.

`/api/search` admite únicamente el método GET y un parámetro de texto como entrada para la búsqueda.

Por último, mediante el método POST sobre `/api/auth/login` proporcionando información sobre el identificador de usuario y la contraseña, se devuelve un JWT para autenticación en peticiones futuras.

Así mismo, se han utilizado los códigos de respuesta de HTTP para indicar el resultado de las operaciones. El código 200 indica que la petición es correcta y se envía la respuesta, y el código 201 que se ha creado un recurso correctamente. Los códigos de error que se utilizan son: 400 para una petición mal formada, 401 para indicar que la operación requiere autenticación que no se ha proporcionado y 404 para peticiones de recursos no encontrados.

# *Lista de Figuras*


1.1. Captura de pantalla de la página de Pocketbeat accedida desde un ordenador de escritorio. . . . . 2
1.2. Captura de pantalla de la página de Pocketbeat accedida desde un ordenador de escritorio. Vista de vídeos. . . . . 2
1.3. Vista inicial. . . . . 3
1.4. Vista de vídeos. . . . . 3

2.1. Captura de pantalla de la aplicación de Boiler Room, versión 3.0.9. . . 8
2.2. Captura de pantalla de la aplicación WAV, versión 1.3. . . . . 9
2.3. Captura de pantalla de la aplicación BE-AT.TV, pantalla inicial. Versión 1.4. . . . . 10
2.4. Captura de pantalla de la aplicación BE-AT.TV, menú lateral. Versión 1.4. . . . . 10

3.1. Pantalla principal aplicación. . . . . 15
3.2. Arquitectura del sistema. . . . . 18
3.3. Arquitectura del sistema de integración continua. . . . . 20
3.4. Arquitectura del sistema con iconos de las tecnologías utilizadas. . . . . 20
3.5. Arquitectura de la infraestructura de integración continua del sistema con iconos de las tecnologías utilizadas. . . . . 23
3.6. Diagrama del proceso, con los diferentes procesos que realiza TeamCity y Octopus. Obtenido de un artículo[27] oficial de JetBrains. . . . . 23

4.1. Diagrama UML de las clases que forman los modelos necesarios en la base de datos. . . . . 29
4.2. Interacción entre cliente y servidor utilizando *tokens*. Imagen obtenida de la página de JWT[31]. . . . . 35
4.3. Creación de un despliegue desde Octopus Deploy. . . . . 35
4.4. Creación de un despliegue desde Octopus Deploy. . . . . 36
4.5. Creación de un despliegue desde Octopus Deploy. . . . . 36

5.1. Diagrama del despliegue en el servidor de prueba. . . . . 42

A.1. Pantalla con el menú desplegable. . . . . 54
A.2. Pantalla inicial tras abrir por primera vez la aplicación. . . . . 55
A.3. Pantalla de inicio de sesión. . . . . 56
A.4. Pantalla de registro. . . . . 57
A.5. Pantalla de búsqueda de artistas y sesiones. . . . . 58
A.6. Pantalla de los vídeos de la aplicación, una vez el usuario se ha identificado o registrado. . . . . 59
A.7. Pantalla con el reproductor, en modo vídeo. Arrastrando el vídeo hacia abajo se minimiza en un reproductor pequeño, el de la figura A.9. De esta forma se puede seguir escuchando la sesión mientras se explora el contenido. . . . . 60
A.8. Pantalla con el reproductor, en modo audio, donde se muestra una imagen estática. De la misma forma que en modo vídeo, se puede minimizar. . . . . 61
A.9. Pantalla con el reproductor minimizado. . . . . 62
A.10. Pantalla de la lista de artistas, se pueden filtrar por inicial seleccionando la letra en la zona superior con fondo blanco. . . . . 63
A.11. Pantalla con la información de un artista. . . . . 64
A.12. Pantalla con los vídeos guardados por el usuario. . . . . 65
A.13. Pantalla con los artistas seguidos. . . . . 66
A.14. Pantalla del perfil del usuario, donde se puede editar la información. No hay transiciones a otras pantallas; el guardado es automático. . . . . 67
A.15. Pantalla con información de soporte. Hay enlaces al correo electrónico y redes sociales de la plataforma. . . . . 68
A.16. Pantalla para funcionalidades sin implementar, como My Stream y Connections. . . . . 69

# *Lista de Tablas*

3.1. Requisitos funcionales de la aplicación móvil. . . . . 14
3.2. Requisitos funcionales del API . . . . . 16
3.3. Requisitos no funcionales del API . . . . . 17

4.1. URIs y recursos . . . . . 27
4.2. Modelos de objetos de transferencia de datos. . . . . 30

6.1. Esfuerzos . . . . . 44

B.1. Operaciones de REST admitidas por cada punto de acceso del API. . . 71