Desarrollo de la app Share WiFi y su infraestructura.

Ricardo Ruiz López
me@ricardoruizlopez.com

Master Thesis at Polytechnic University of Valencia.
Master in Software Engineering and Formal Methods.

Septiembre 2014

Director:
Francisco Javier Jaén Martínez
fjaen@upvnet.upv.es

ABSTRACT

Hoy en día la industria de las aplicaciones para dispositivos móviles están en su máximo apogeo. También, y gracias en parte a las aplicaciones móviles, el sector de la economía colaborativa o economía p2p (sharing economy) está en una etapa boyante. Tenemos ejemplos como Uber, Blablacar o AirBnb.

Esta tesina industrial trata de realizar una aplicación que usa las ideas de la economía colaborativa para compartir los puntos WiFi que tenemos en nuestras casas y muchas veces están infrautilizados.

Para realizar con éxito este proyecto, no sólo es necesario crear la correspondiente app, sino toda una infraestructura detrás, conocida comúnmente como backend, la cual muchas veces es incluso más costosa de realizar que la app.

Estos backend tradicionalmente se han hecho con Java o PHP y se ejecutan en contenedores de aplicaciones empresariales, y sin tener demasiado en cuenta futuros problemas de escalabilidad.

Una posibilidad de reducir dramáticamente los tiempos de desarrollo de backends es usar los recientemente aparecidos mobile backend as a service (MBaas) o simplemente BaaS.

MBaaS nos llevan un paso más allá, elevando el nivel de abstracción para centrarse en la lógica de app obviando detalles técnicos y encargándose de aspectos técnicos como la escalabilidad y seguridad de manera automática.

Normalmente los MBaaS nos permiten guardar datos, enviar notificaciones push, añadir porciones de código para crear lógica, integración con redes sociales, analizar los patrones de uso de los usuarios (estadísticas)... a veces incluso SDKs específicos para iOS o para Android que todavía simplifican más nuestros desarrollos en lugar de usar REST o similar.

Este proyecto está completamente hecho usando uno los BaaS más populares que hay hoy en día, Parse.

AGRADECIMIENTOS

Agradezco la enorme ayuda recibida por gente desconocida de todas partes del mundo a través los foros de Stack Overflow, los foros de desarrolladores de Apple y los foros de Parse.

También agradezco todos los diseños de la app hechos por mi compañera de trabajo Jessica (http://cargocollective.com/jssicajohnson).

Table of Contents

1. Introducción.

1.1. Objetivos.

Los objetivos de esta tesina son, por una parte crear una aplicación con su infraestructura totalmente funcional y lista para ser explotada en el mercado. Y por otra parte, aprender sobre los Backend as a Service (BaaS), cuyo conocimiento dispara la productividad y rebaja drásticamente el coste en los proyectos de aplicaciones móviles, ya que de alguna u otra manera, casi todas las apps hoy en día requieren de Internet o infraestructura de alguna u otra forma.

1.1.1. Monetización.

Este proyecto tendrá unos gastos fijos en lo que a servidores se refiere. Podría ponerse a la venta en la App Store por un dólar (o similar), pero ya que se procurará que llegue al mayor número posible de usuarios, es conveniente que sea totalmente gratuito. Es evidente que una aplicación gratuita será descargada muchas más veces que una de pago. Por ello, para procurar o intentar financiar los gastos fijos del proyecto, se insertará un pequeño anuncio en la vista principal del programa y otro anuncio a pantalla completa (intersitial) en una de las partes clave del programa; justo después de enviar un mensaje a otro usuario.

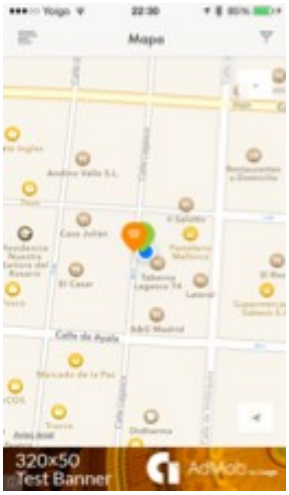


Figura 1. Anuncio pequeño en la vista principal de la app.

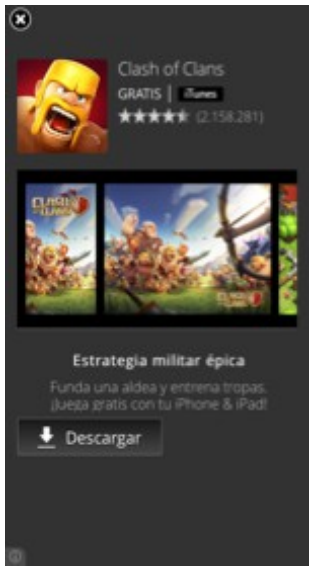


Figura 2. Anuncio intersitial (pantalla completa) visto después de enviar un mensaje.

Se ha elegido AdMob, el sistema de anuncios de Google, en lugar de iAd. El motivo es que los intersitial, que son los que realmente dan dinero, en Google tienen un fillrate mucho más elevado que iAd.

1.2. Aplicaciones.

Las aplicaciones son claras gracias a la economía colaborativa. Básicamente, optimizar recursos, en este caso el caudal que tenemos de Internet, de forma que podemos compartir (ya sea gratis o compartiendo gastos) lo que nos sobra. Y además otras personas disfrutan de Internet por precios ridículos.
De hecho, las grandes empresas de telecomunicaciones ya están usando algo similar para aliviar las sobresaturadas redes móviles 3G y 4G. A este idea se le llama offloading [1], [2] y.[3]
Y en época de crisis como la actual, los negocios que ayudan a ahorrar, tienen más mercado.

1.3. Justificación.

Podemos justificar el desarrollo de esta tesina desde varios puntos de vista.

1.3.1. Economía colaborativa.

No hay duda que hoy en día la economía colaborativa está en su apogeo. Tenemos ejemplos como Uber o AirBnb. Básicamente consiste en compartir (gratuitamente o por dinero) los recursos que nos sobran, es decir, están mal gastados u ociosos. Por ejemplo podemos alquilar nuestro coche cuando no lo usamos, o podemos alquilar nuestra plaza privada de garaje en las horas que estamos trabajando y por tanto estamos fuera.
Sus beneficios evidentes son: Ahorrar dinero, suele ser bueno para el medio ambiente, da flexibilidad a la hora de gestionar los recursos y es práctico.

En la siguiente figura podemos ver diferentes apps que cubren diferentes nichos de mercado.

Figura 3. Apps y nichos enla economía colaborativa [4].

1.3.2. El mercado de smartphones.

Hoy en día casi todo el mundo tiene un smartphone en los países desarrollados, lo que abre la oportunidad de ejecutar ricas aplicaciones. El smartphone es la clave para crear nuevos modelos de negocio a través de las apps que son vendidas o distribuidas en las tiendas de aplicaciones.

Ejemplo de índice de penetración de los smartphones en la siguiente figura.

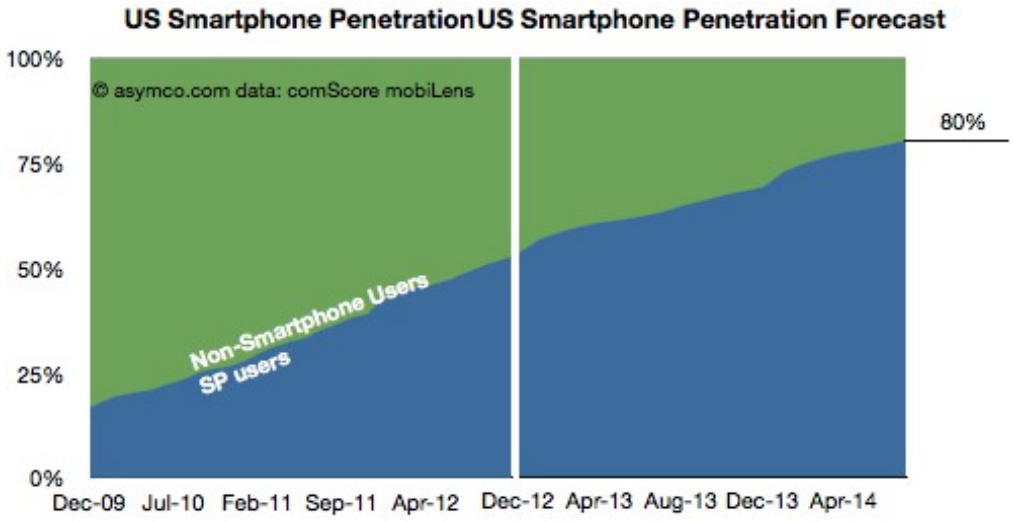


Figura 4. Indice de penetración de smartphones en EEUU, [5].

1.3.3. El mercado de las aplicaciones para móviles.

Una aplicación hecha expresamente y distribuida a través de una tienda de aplicaciones es la pieza clave para que funcione una economía colaborativa, para que personas desconocidas y que a veces viven a miles de kilómetros, puedan ponerse en contacto.
Indiscutiblemente hoy en día el mercado de los smartphones está prácticamente dividido entre dos sistemas operativos: iOS de Apple y Android de Google.

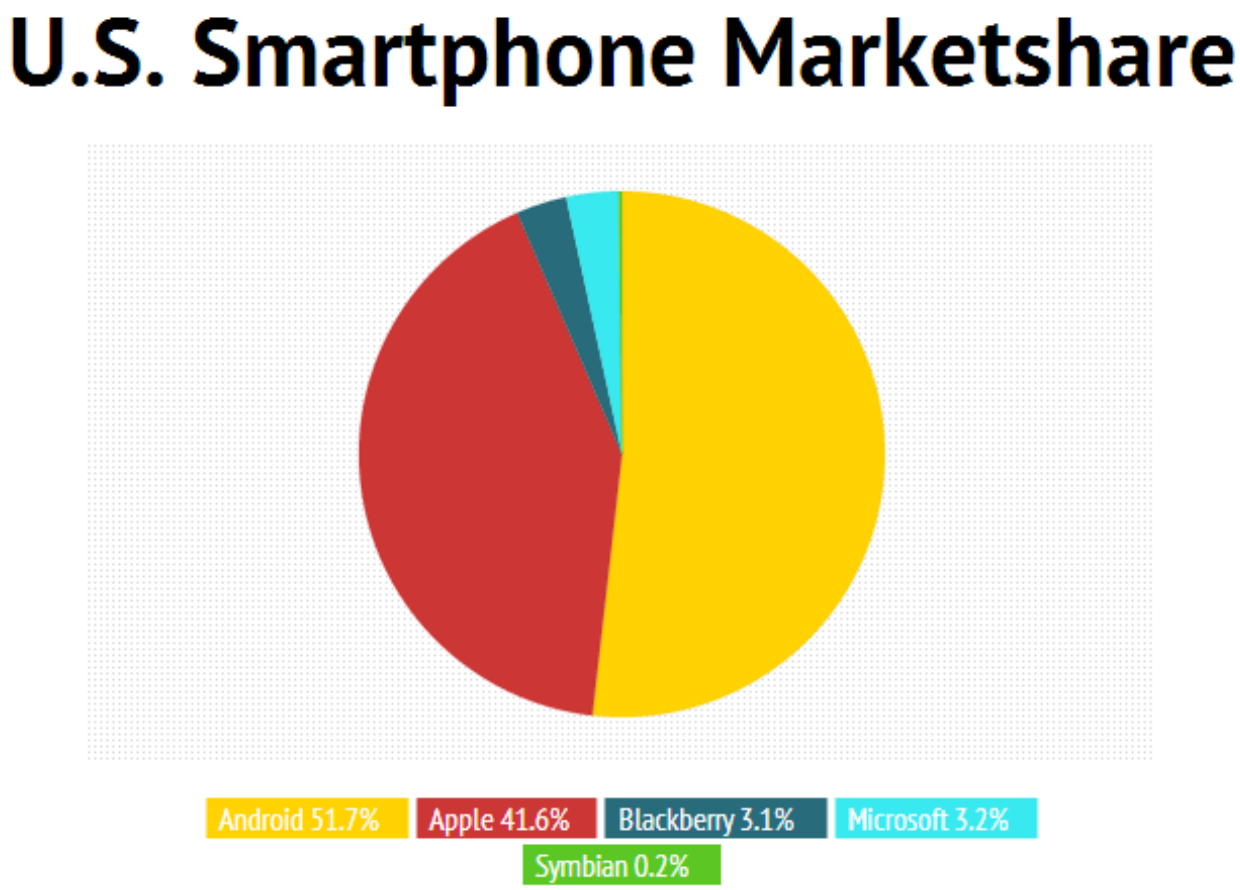


Figura 5. Distribución de los SO móviles en EEUU, [6].

Por ello, para que un modelo de negocio funcione, sería necesario tener una aplicación en el menos las dos plataformas más famosas (y que copan casi todo el mercado), es decir, una app para Android y otra para iOS (iPhone y opcionalmente iPad). Sin embargo, como el que escribe esta tesina sólo es diestro haciendo apps para iOS, sólo hará la versión de iOS por el momento.

1.4. Cuestiones legales.

La aplicación Uber a provocado fuertes protestas en algunas ciudades (como Madrid) por parte de taxistas que alegan que hace competencia desleal, [7].

La aplicación AirBnb también ha creado fuertes protestas entre los hoteleros de todo el mundo, y de hecho ha sido expresamente prohibida en algunas ciudades, como por ejemplo Nueva York, [8].

Normalmente, los modelos de negocios disruptivos, muchas veces quitan un buen trozo del mercado de los antiguos negocios que cubrían ese mercado, y ellos intentan defenderse usando la legalidad o no. Por ejemplo los taxistas alegan que los conductores de Uber no pagan las costosas licencias que los taxistas sí pagan.

Algo similar ocurre con las compañías telefónicas que en sus contratos explícitamente prohiben el poder compartir Internet. Sin embargo, mientras no haya beneficio propio (ánimo de lucro) y sólo se comparta gastos, no parece que haya problema tal y como dice la CMT, [9].

Otro punto legal sería la identificación de una dirección IP cuando hay un acto delictivo. Parece que hoy en día tampoco es un problema, [10].

2. Estado del arte.

Este capítulo trata de lo más actual o la tecnología de vanguardia en lo que a la tesina se refiere.

2.1. iPhone.

iPhone es una línea de teléfonos inteligentes diseñado y comercializado por Apple Inc. Ejecuta el sistema operativo móvil iOS antes conocido como "iPhone OS". Steve Jobs anunció el iPhone en 2007, tras varios rumores y especulaciones que circulaban desde hacía meses. El iPhone se introdujo inicialmente en los Estados Unidos el 29 de junio de 2007. Fue nombrado «Invento del año» por la revista Time en el 2009.

Las ventas de iPhone han hecho de Apple una de las empresas más ricas del planeta. En el último trimestre de 2013 se vendieron 51 millones de iPhones, superando el récord del trimestre del año anterior de 47,8 millones. En Marzo de 2014 las ventas acumuladas eran de 500 millones de iPhone en todo el mundo, [11] y [12].

El iPhone es el precursor de los smartphones actuales, y basa su funcionamiento en 4 puntos, que han ido evolucionando y mejorando con las diferentes generaciones (8 hasta el momento).

-Pantalla: Evidentemente al ser más pequeño que un ordenador de sobremesa hay algunas limitaciones. Se ha eliminado el concepto de multiventana, pero eso no es problema porque las apps usan otros paradigmas para interactuar con el usuario.

La pantalla y el sistema gráfico usa OpenGL ES (versión para sistemas empotrados) con lo que es capaz de ejecutar gráficos 3D en tiempo real. Esto es usado tanto por el sistema operativo, como escritorio así como los videojuegos.

-Recursos limitados: Aunque los recursos han ido creciendo de manera espectacular en las últimas generaciones (incluso procesadores con varios núcleos), no hay que olvidar que es un dispositivo portátil y limitado por una batería, y esto hay que tenerlo en cuenta a la hora de desarrollar las apps. Además, el SO ejecuta las apps en un entorno controlado llamado sandbox, de forma que lo que

puede hacer las apps está limitado (por ejemplo, salvo excepciones como apps de VOIP, las apps no se pueden quedar en background indefinidamente gastando batería).

-Conectividad: WiFi, Bluetooth, 3G, 4G, e incluso NFC para pagos en el último modelo iPhone 6. También Bluetooth LE para conectarse con relojes, podómetros y diversos instrumentos. Hoy en día un smartphone sin una conexión permanente a Internet es prácticamente inútil.

-Entrada y salida: El iPhone tiene diferentes sensores que se pueden usar para diversas tareas. Acelerómetro, GPS, giroscopio, sensor de presencia, lector de huella, micrófono, altavoz, pantalla multitáctil (con hasta 5 toques a la vez), cámara de fotos, de vídeo (tanto normal como cámara lenta), vibración... con todo este sistema de entrada/salida podemos realizar las apps más variopintas imaginables.

2.2. Xcode.

Xcode es el entorno de trabajo que nos propone Apple para crear aplicaciones para iOS. Es un conjunto de herramientas integradas:

- Xcode IDE: Integra en una misma interfaz diversas tecnologías como editor de código fuente, depurador, editor de interfaces gráficos de usuario, gestión de los repositorios de código (git, svn). Live Issues te avisa inmediatamente de cuando hay un error en tu código e incluso te ofrece sugerencias. También incluye integración con el portal de desarrolladores de iPhone, de forma que todo lo relaciona con certificados, claves privadas y claves públicas lo facilita.

- Compilador LLVM: El compilador creado por Apple para Objective-C. Además de compilar, también analiza tu código (static analyzer) buscando fallos semánticos, como por ejemplo uso de variables sin inicializar o por ejemplo uso incorrecto de la gestión de memoria manual (retain/release deben estar balanceados).

- Instruments (Profiler): Es el profiler de Apple que sirve entre otras cosas para tomar mediciones de nuestra aplicación mientras se ejecuta en el dispositivo. Podemos buscar cuellos de botella en el código y saber exactamente donde debemos optimizar nuestro código. También se usa para revisar la gestión de la memoria. Ambos puntos, tanto la memoria como la velocidad de ejecución son especialmente críticos en un dispositivo portátil (o sistema empotrado con limitaciones técnicas) por lo que no debemos dejar de usar esta herramienta en el desarrollo de nuestras apps.

- Simulador: De la misma forma que ejecutamos nuestra app en nuestro móvil a través de un cable, también podemos usar un simulador, el cual podemos configurar cualquiera de los iPhones disponibles (con diferentes tamaños de pantalla) para probar nuestra aplicación. Aunque tiene alguna limitación, como por ejemplo que carece de cámara o de acelerómetro/giroscopio, normalmente usamos el simulador ya que es más rápido el proceso que conectar el iPhone. También tiene otras ventajas como que podemos simular posiciones del GPS en cualquier parte del mundo, incluso podemos simular que nos movemos por el mundo y crear rutas.

2.3. El sistema operativo iOS.

iOS es el sistema operativo creado por Apple para iPhone, iPad y iPod Touch. Es una variante de Mac OS X adaptada a sistemas embebidos. Usa un núcleo basado en Match, hereda parte de las tecnologías de la empresa NeXT (que fue comprada por Apple) y utiliza los frameworks Cocoa (Cocoa Touch).

Se compone de 4 capas de abstracción:

-Core OS (núcleo del sistema operativo): Todo lo relacionado con memoria virtual, sistemas de ficheros, sockets (TCP/IP), seguridad, gestión de memoria, procesos, planificador de procesos.

-Core Services (Servicios principales): Servicios como conexiones de redes, acceso a ficheros y bases de datos, acceso a agendas y a hilos.

-Media (Medios de comunicación): Diversos frameworks y librerías que permiten construir aplicaciones con gráficos avanzados, reproducción de sonido y vídeo, gráficos 3D en tiempo real...

-Cocoa/Cocoa Touch: Conjunto de herramientas que permiten acceder a tipos de datos básicos así como control de eventos.

En general el lenguaje de programación de aplicaciones para iOS es Objective-C, aunque nada impide usar C o C++ (muchos juegos se hacen en esos lenguajes). Y ahora mismo Apple está trabajando en un nuevo lenguaje llamado Swift.

2.4. El lenguaje Objective-C.

El lenguaje Objective-C fue creado en 1980 y es una extensión del lenguaje C. Añade a C la programación orientada a objetos. Este lenguaje se usa principalmente en los productos de Apple, tanto en su SO para móviles iOS (iPod Touch, iPhone, iPad) como para ordenadores de escritorio (Mac OS X).

Es un lenguaje que suele ser más “verbose” y por tanto fácil de leer. Por ejemplo comparar esta línea de ObjC:

```
[myColor isEqualToString:@”Red”];
```

con su homóloga en C:

```
strcmp(myColor, “Red”);
```

En C es más corto la sentencia, pero más difícil de leer.

Además de esto, ya que ObjC es una capa encima de C, siempre se podría usar C si el programador lo desea.

Modelo-Vista-Controlador.

Es el patrón de diseño estrella en iPhone/ObjC/Xcode. Divide la app en tres partes diferenciadas:

-Modelo: Contiene todos los datos de la app. No define de ninguna manera el comportamiento de la app.

-Vistas: Es la interfaz gráfica del usuario (botones, imágenes, campos, etiquetas, animaciones...) que sirve para mostrar datos al usuario y también para coger información del usuario.

-Controlador: Son los que guardan la lógica de la app. Es responsable de recibir la información del usuario, procesarla, guardarla en el modelo si fuera necesario, aplicarle la lógica que quiera y volver a presentarla al usuario.

En realidad no es un patrón de diseño, sino que es un conjunto de patrones como se ve en la siguiente figura.

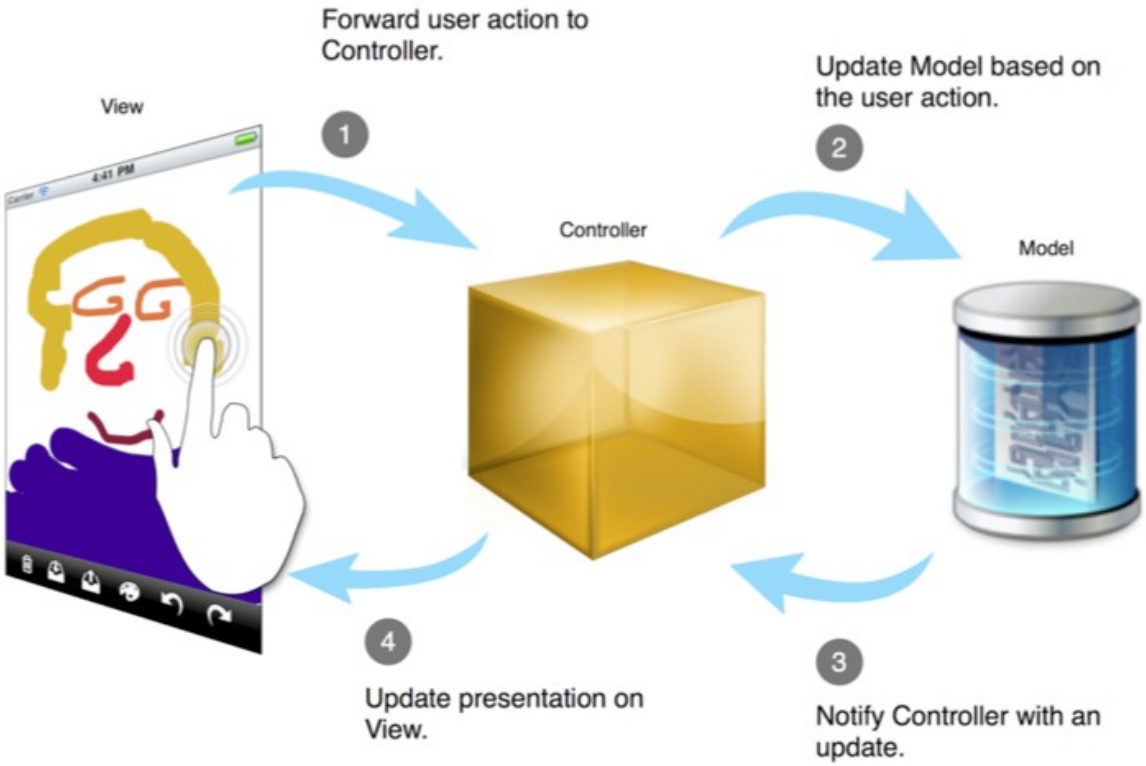


Figura 6. Ejemplo de Modelo-Vista-Controlador, [13].

Composite: Las vistas forman una jerarquía y cada una de las vistas pueden interactuar con el usuario.

Command pattern: Los objetos de las vistas ejecutan código que se encuentra en otros objetos como los controladores.

Mediator: El controlado actúa como middleman o intermediario entre las vistas y el modelo formando un flujo bidireccional de datos entre ambos.

Strategy: Un controlador puede ser un “strategy” para una vista cualquiera. Una vista de aisla de forma que sólo se usa para mostrar datos y delega al controlador todas las decisiones sobre los datos.

Observer: Un objeto modelo puede informar a quien lo desee de cambios en sus datos usando este patrón. También es común en iOS usar el patrón Delegation para esta tarea.

2.5. Mobile Backend as a Service.

El mercado de las aplicaciones móviles es extremadamente dinámico y cambiante, con ciclos de vida muy cortos. Es por ello que cualquier herramienta, paradigma o metodología que ayuda a reducir estos tiempos, debe ser bienvenida.
Tradicionalmente la infraestructura de las apps se suele hacer creando aplicaciones empotradas en Java Empresarial, o usando PHP, e intercomunicando estos servicios usando servicios web como SOAP o REST, que proporcionan los datos a las apps a través de JSON o de XML. También estos programas en servidores tienen que conectarse a bases de datos, a veces relacionales y a veces NoSQL (o una aproximación híbrida como suele ser en muchos juegos sociales). Además de ello debido a la peculiaridad de que hay millones de dispositivos móviles, podría necesitarse una alta demanda y el sistema no estar preparado, y para cuando hubiera sido actualizado, la aplicación ya hubiera muerto de éxito (ya que los ciclos de vida de las apps son muy pequeños). Todas estas infraestructuras son costosas de hacer en tiempo, en dinero y en mantenimiento.

Los MbaaS [14], o simeplemente BaaS, Backend as a service, es una nueva forma para proveer de nuestras apps de servicios en la nube. Principalmente todo lo referido a bases de datos, almacenamiento de ficheros, notificaciones push, login con redes sociales, lógica, estadísticas...
No existe un estándar a día de hoy por lo que cada fabricante utiliza una forma diferente de acceder a estos servicios. Por ejemplo, uno de los más famosos es Parse, el cual proporciona librerías para las más importantes plataformas (iOS y Android entre otros) de tal forma que se eleva el nivel de abstracción, ya que no hay que lidiar con conexiones de red, servicios web, REST, SOAP, reintentos, mapeos de datos... todo eso está abstraído por las librerías proporcionadas lo que permite desarrollar más rápido. En el caso concreto de Parse, se puede expresar lógica en la parte del servidor usando Javascript. Además, se paga por lo que se usa y escala automáticamente.

Los BaaS han creado un nuevo mercado que se calcula en 7700 millones de dólares para el 2017, [15].

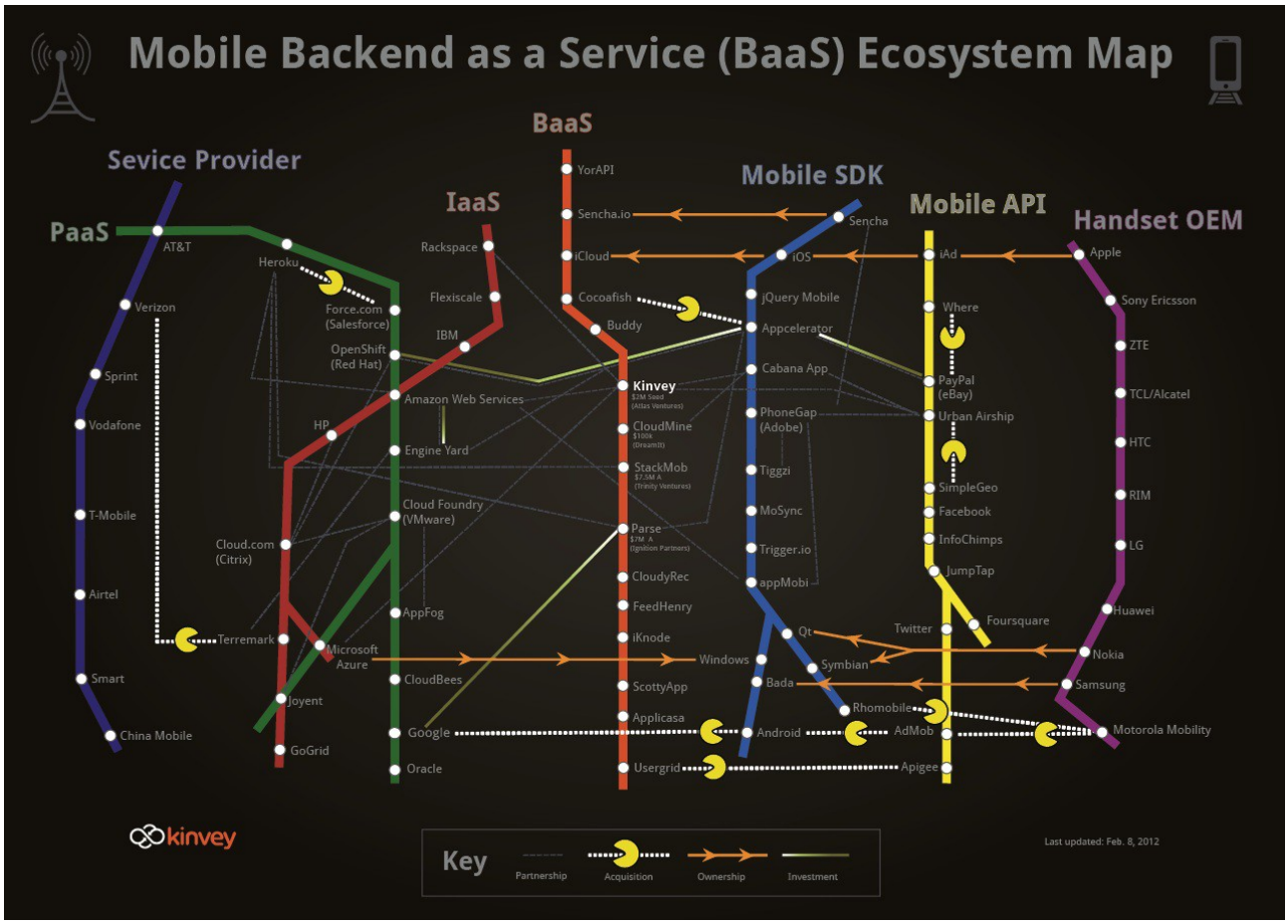


Figura 7: Infografía del mercado de BaaS y sus adquisiciones, [16].

Sin embargo, no es oro todo lo que reduce. Al usar un BaaS normalmente nos volvemos dependientes de ese fabricante, porque no hay estándares industriales, y podría darse el caso de que la empresa quiebre o por alguna razón nos apague nuestra app hospedada. Un ejemplo lo tenemos en Stackmob, que era un BaaS comprado por PayPal. Al cabo de unos meses de hacer la compra, PayPal decidió cerrarlo, [17]. Además, cada fabricante tiene a nivel técnico sus pros y sus contras que deben ser analizados individualmente.

3. Análisis y modelado del sistema.

3.1. Especificación de requisitos.

Existen muchas formas de especificar requisitos,como por ejemplo casos de uso, historias de usuario, un simple listado de requisitos... a veces se puede hacer más formas y detallado y otras veces más informal y menos detallado.
En el mundo del desarrollo móvil, el ciclo de vida las aplicaciones (tanto el desarrollo como explotación) puede ser muy corto, a veces incluso de semanas. Por tanto hay que estar preparado para algo cambiante y ofrezca flexibilidad, ya que de hecho a medida que rápidamente se desarrolla la app, se verán nuevos requisitos y se cambiaran los que se estaban realizando. Por eso para expresar los requisitos en esta tesina lo haré a modo de lista de funcionalidades a modo de historia de usuario. No hay que olvidar que este proyecto ha sido llevado en sólo unas pocas semanas.

3.1.1. Registro y uso anónimo.

La aplicación tiene ciertas funcionalidades que requieren obligatoriamente un usuario registrado y logeado. De la misma forma que tiene otras funcionalidades para lo cual no es necesario estar registrado, es decir, se pueden usar anónimamente. Es importante notar que obligar al usuario registrarse nada más arrancar la app por primera vez, es algo que espanta a posibles usuarios, por tanto se pedirá registro cuando sea estrictamente necesario y lo más tarde posible.

3.1.2. Ver los puntos WiFi en el mapa.

Es la funcionalidad más importante de la aplicación. Se muestra en la pantalla principal un mapa con todo los puntos WiFi (hotspots) que hay disponibles. Hay 3 tipos y cada uno tiene un color diferente. De gastos compartidos, gratuito, desconocido. Se puede filtrar por tipo de punto. También se puede cambiar el tipo de mapa a estándar, satélite o híbrido. Además se puede ver en el mapa un punto azul que indica tu localización actual y un botón para ir a ella. No requiere logeo.

3.1.3. Ver detalle de cada punto.

El detalle de un punto contiene en nombre de la red (SSID), tipo de punto (gastos compartidos, gratis, desconocido), dirección postal del punto, dueño del punto, descripción del punto (donde el dueño comenta las condiciones), valoración promedia del punto en forma de estrellitas (entre 0 y 5), listado con las valoraciones de la gente, botón para enviar un mensaje al dueño del punto, botón para ver el perfil del dueño, botón para valorar el punto, botón para denunciar el punto. Ver esa información no requiere logeo, pero realizar ciertas acciones requiere logeo (enviar mensaje, denunciar, valorar).

3.1.4. Enviar mensaje al dueño de un punto.

Posibilidad de enviar un mensaje al dueño del punto para poder ponerse en contacto y hacer el intercambio del recurso (el punto WiFi). Al pulsar enviar, el destinatario recibirá una notificación push avisándole, ade8más de verlo en su bandeja de entrada y con el número de mensajes no leídos tanto en el badge de la app como del menú. Requiere logeo.

3.1.5. Valorar punto WiFi.

Se valora con un comentario que es opcional y con un determinado número de estrellitas (de 0 a 5). No se puede valorar más de una vez. Si se hace una segunda vez, se sobreescribe la valoración y se avisa al usuario que valora que ya había valorado.

Al valorar, se envía una notificación push al dueño del punto valorado avisándole.
Un usuario no puede valorar sus propios puntos WiFi. Requiere logeo. Requiere estar cerca del punto, dentro de un radio de un kilómetro.

3.1.6. Valorar un usuario.

Se puede valorar un usuario con el que previamente hemos tenido algún contacto por mensajería (nos ha contestado algún mensaje). Se valora con un comentario (opcional) y un sistema de 5 estrellas donde elegimos desde 0 a 5. Requiere logeo. Hay que tener en cuenta el caso en que el receptor haya borrado un mensaje, por eso los mensajes no se borran, sino que se marcan como borrados.

3.1.7. Dar de alta un punto WiFi.

Se puede hacer desde dos sitios diferentes. Desde la vista principal del mapa, en donde vemos nuestra posición actual con un punto azul, pulsando ahí. También se puede hacer desde nuestro propio perfil. Es necesario introducir en nombre del punto (SSID), dirección postal, tipo de punto (gastos compartidos o gratuito). Requiere logeo.

3.1.8. Ver mi propio perfil de usuario.

Una vez logeado, se puede ver el perfil propio donde podemos cambiar nuestra foto, ver las valoraciones dadas y recibidas, crear un punto WiFi y ver todos los puntos WiFi que he creado. Requiere logeo.

3.1.9. Ver listado de puntos WiFi que el usuario logeado ha creado.

Desde el perfil de usuario puedes ver un listado de los puntos que has creado y borrarlos (left swipe). Posibilidad de editar el tipo (pago/gratis), dirección, nombre, descripción. Posibilidad de borrar el punto (así como sus comentarios asociados y valoraciones). Requiere logeo.

3.1.10. Ver reseñas dadas y recibidas.

Desde el propio perfil de usuario se pueden ver todas las reseñas que me han puesto y todas las reseñas que he puesto yo a otros usuarios. Tanto el comentario como la valoración (estrellitas), así como el nombre de quien da la reseña o la recibe y la fecha. Requiere logeo.

3.1.11. Ver mensajes recibidos.

Los usuarios logeados tienen una bandeja de entrada donde se muestran los mensajes recibidos. Los mensajes no leídos se marcan con un color especial. También se marca el número de mensajes no leídos en el badge de la aplicación. Pulsando un mensaje se puede ver el detalle del mensaje y también se puede contestar a la persona que envió el mensaje. Requiere logeo.

3.1.12. Denunciar un punto.

Desde el detalle del punto WiFi, se puede denunciar. Se escribe un comentario y se envía la información de ese comentario así como el identificador del punto y el identificador del usuario que lo envía. Una vez llegado esa información al servidor, se reenviará al administrador del sistema. Requiere logeo.

3.1.13. Denunciar un usuario.

Desde el detalle de un punto se puede ver el usuario que lo creo,y desde ahí el detalle de un usuario

cualquiera.Se puede denunciar un usuario con un comentario. El identificador de ambos usuarios y el comentario será enviado al servidor para que lo reenvíe al administrador del sistema. Requiere logeo.

3.1.14. Publicidad AdMob.

En el mapa principal habrá un pequeño anuncio y a la hora de enviar un mensaje, justo al ser enviado con éxito se mostrará un intersitial (anuncio a pantalla completa).

3.1.15. Perfil de un usuario cualquiera.

Desde el detalle de un punto WiFi, puede verse su dueño. Pulsándolo se puede ver el perfil público de un usuario cualquiera donde vemos todas las valoraciones y comentarios que la gente le ha dejado, así como el promedio de las valoraciones y la foto del usuario si la tuviera. Desde esa ventana también podemos denunciar al usuario. No se puede denunciar un usuario a sí mismo. Requiere logeo.

3.1.16. Formulario de contacto.

La aplicación da la posibilidad de ponerse en contacto con el administrador del sistema para lo que quiera, para decir sugerencias o fallos. Se puede hacer tanto logeado como anónimamente. Si es anónimamente, se pide el correo electrónico.

3.1.17. Ayuda.

La aplicación tiene una sección de ayuda donde explica y da consejos sobre como compartir WiFi, por ejemplo aconsejando que no es conveniente hacer cosas críticas como operaciones bancarias o ver correo porque pueden estar leyendo los paquetes de red, o consejos de usar una VPN...

3.2. Modelado conceptual.

Para dar soporte a todos los requisitos planteados en el sistema, necesitamos las siguientes entidades y relaciones:
Usuario: Contiene su nombre de usuario, la forma con la que ha iniciado sesión (usando alguna red social) y foto.

- Hotspot: Contiene el nombre de la red (SSID), descripción (donde el usuario pone las condiciones para usar ese punto WiFi), dirección postal y tipo de punto. Se relaciona con un Usuario para saber quien es su creador. También se relaciona con las entidades HotspotRating para saber las valoraciones que ha recibido ese hotspot.

- Message: Contiene un mensaje, el flag para saber si está borrado o no, el flag para saber si ha sido leído o no, y se relaciona con la entidad usuario,una para saber quien es el receptor y otra para saber quien es el emisor.

- UserRating: Contiene una valoración de un Usuario, es decir, un determinado número de estrellitas, de 0 a 5, así como un comentario. Está relacionado con un Usuario que es quien ha creado la valoración y con otro usuario que es quien recibe la valoración.

- HotspotRating: Es la valoración de un hotspot representada como el número de estrellitas (de 0 a 5) y un comentario opcional. Se relaciona con un Hotspot que es el que recibe la valoración y tambien con un Usuario que es quien escribe la valoración.

- Installation: Posiblemente esta sea la menos intuitiva. Para poder enviar una notificación push, es necesario conocer el device token de un dispositivo, y a su vez, hay que asociar a un usuario con uno o varios dispositivos (Instalaciones). Un Usuario puede tener varios dispositivos y una Installation se relaciona con uno o cero Usuarios.

A continuación podemos ver el modelo de datos conceptual.

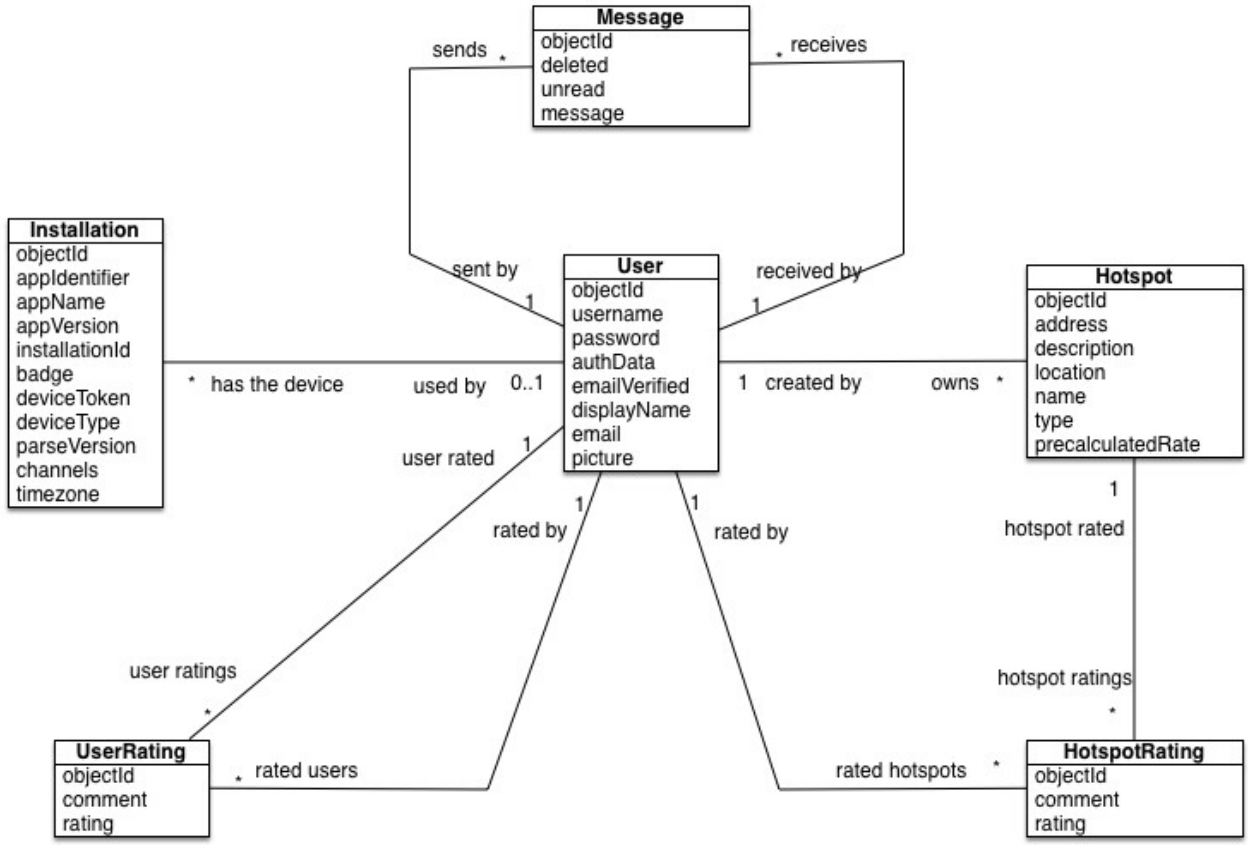


Figura 8. Modelo de datos conceptual.

4. Diseño y arquitectura del software.

4.1. Sistemas BaaS.

Para la realización de la infraestructura, se ha elegido un BaaS para acortar los tiempos de desarrollo lo máximo posible. Hoy en día existen decenas de sistemas BaaS, entre ellos algunos muy conocidos son: Parse, Backbeam, Kinvey, Backendless, Kony... se puede ver una comparación

de las funcionalidades en la siguiente tabla.

	Parse	BackBeam	Kinvey	Backendless	Kony
Lógica en servidor	V	V	V	V	V
Base de datos	V	V	V	V	V
Notificaciones Push	V	V	V	V	V
Logeo redes sociales	V	V	V	V	V
Versionado APIs	X	X	X	V	V
Migración datos	X	X	X	X	X
Almacenamiento archivos	V	V	V	V	V
Librería iOS/Android	V	V	V	V	V

Tabla 1. Comparación de diferentes sistemas BaaS.

Como puede apreciarse, prácticamente hoy en día todos los BaaS ofrecen más o menos lo mismo, por lo que una razón para elegir uno u otro importante es quien hay detrás, ya que en caso de que desaparezca, adaptar nuestra app a otro sistema llevaría tiempo por tanto es un riesgo que hay que calcular.

Para este trabajo he elegido uno de los sistemas más populares, Parse. El motivo es principalmente quien hay detrás. En Abril del 2013 Facebook compró Parse en un acuerdo multimillonario, [18]. Por ello su continuidad está prácticamente asegurada. No hay que olvidar el caso de Stackmob comentado anteriormente en el punto “Estado del arte/MbaaS”.

4.2. Modelo de datos.

En general los sistemas BaaS suelen utilizar bases de datos NoSQL (ya que están mejor preparadas para la escalabilidad) y por ello cada sistema BaaS suele tener un sistema peculiar para realizar la base de datos. Una de las desventajas de Parse es que carece de un diagramador donde podamos ver todas las entidades y sus relaciones de golpe. Tenemos una especie de consola donde vamos creando las entidades, añadimos los atributos y las relaciones. Hay que notar que esa es una forma, otra forma es de manera implícita, a medida que vamos solicitando atributos o entidades o relaciones desde las aplicaciones por primera vez. Aunque esto puede ser peligroso ya que si cometemos una errata en un nombre desde el código de la app, podríamos crear cosas que no queremos. De hecho Parse tiene la opción en su consola de bloquear la creación de entidades o clases desde las aplicaciones.

Ejemplo de la consola con las entidades/clases necesarias.

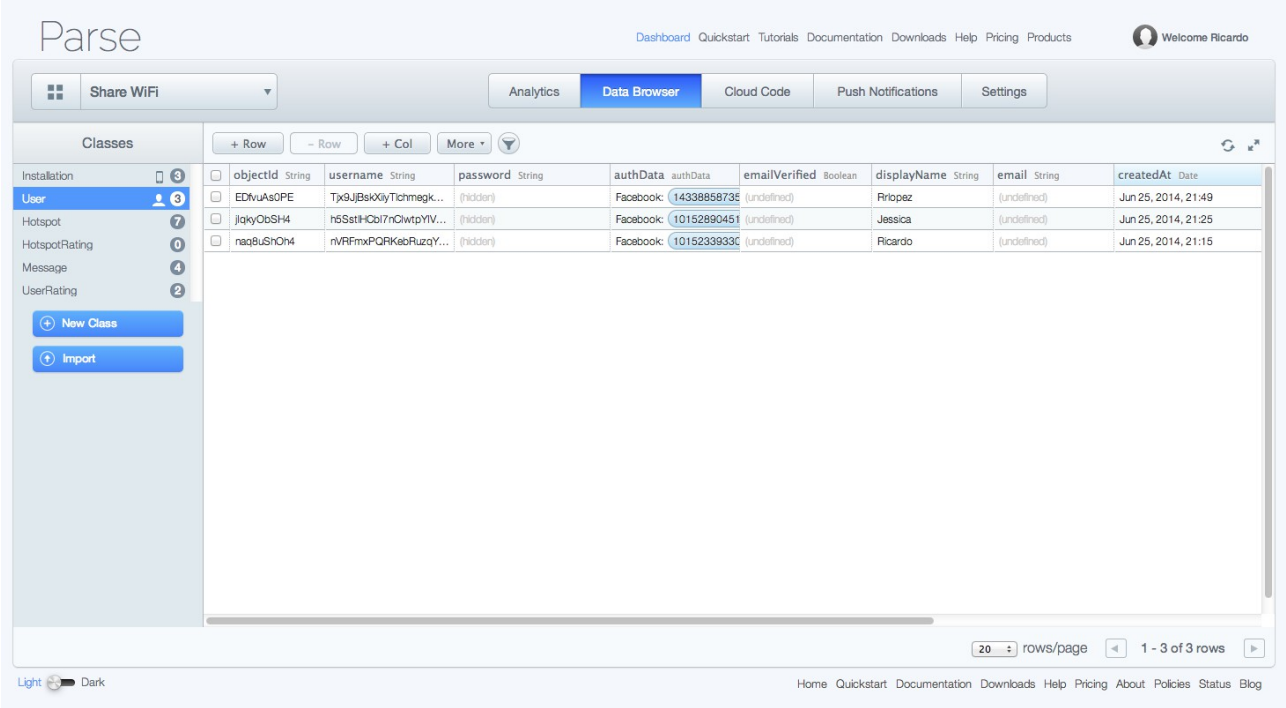


Figura 9. Ejemplo de la consola de Parse con la base de datos.

También es de notar que hay cuatro formas de crear relaciones ([19]) que es necesario conocer para aplicar la que mejor toque en todo caso:

- Pointers. Para relaciones de uno-a-uno o de uno-a-muchos. Según Parse debemos usar este tipo de relación cuando potencialmente pueden haber muchos objetos relaciones (más de 100). En caso contrario (pocos objetos), es más conveniente usar Arrays. Se usado en la app para montar todas las relaciones, ya que potencialmente todas las relaciones podrían contener centenares de objetos.

- Arrays. Para relaciones de uno-a-uno o de uno-a-muchos. Como Pointers pero debemos usarlo cuando hay pocos objetos relaciones implicados. En ninguna parte de la app se ha usado. Se podría usar para por ejemplo relaciones un coche con un color o un personaje con armaduras, ya que se sabe de antemano lo grande que podría ser la relación.

- Parse Relations. Para relaciones de mucho-a-muchos. No se han usado en la app.

- Join Tables. Para relaciones de mucho-a-muchos. Lo mismo que lo anterior pero adeás se quiere añadir algo de información extra a la relación, por ejemplo si modelamos la relación que vemos en Twitter de followed/followers, también es interesante cuando (la fecha) alguien comenzó a seguir a esa otra persona. No se usan en la app.

4.3. Servicios requeridos.

Una de las funcionalidades de Parse es la posibilidad de ejecutar lógica en la parte del servidor. Para este proyecto necesitamos poder realizar estas funciones (incluyo a modo de ejemplo el código de algunos en Javascript/Cloud Code. Aproximadamente todas las funciones tienen la misma complejidad):

-Al borrar un hotspot, debemos también borrar todas sus valoraciones (notar que en la base de datos de Parse no hay reglas de borrado, no se puede hacer por ejemplo un borrado en cascada).

-Un mensaje tiene que tener un origen, destinatario y mensaje obligatoriamente (notar que Parse en los atributos de los datos no se pueden marcar como no nulos u obligatorios).

-Enviar notificación push al destinatario de un mensaje cada vez que se crea el mensaje.

-Función sendHotspotReport que recibe el id de un hotspot, el usuario logeado y un mensaje, para entonces enviar un correo al administrador del sistema.

-Función sendUserReport que recibe el id de un usuario, el usuario logeado y un mensaje, para entonces enviar un correo al administrador del sistema.

-Función sendContactForm. Recibe un id deusuario o email además de un mensaje y lo envía por mail al administrador del sistema.

```
/*
 Sends a report in form of email to the administrator email
 */
Parse.Cloud.define("sendContactForm", function(request, response) {

    // there must be a email
    if (!request.params.email) {
        response.error("You must provide an email.");
        return;
    }

    // there must be a name
    if (!request.params.name) {
        response.error("You must provide a name.");
        return;
    }

    // there must be a subject
    if (!request.params.subject) {
        response.error("You must provide a subject.");
        return;
    }

    // there must be a subject
    if (!request.params.message) {
        response.error("You must provide a message.");
        return;
    }

    // send the email to the administrator
    var message;
    if (request.user) {
        message = "Registered user: " + request.user.get("displayName") + " (" + request.user.id + ")\nName in form: " + request.params.name +
"\nMessage: " + request.params.message;
    } else {
        message = "Anonymous user.\nName in form: " + request.params.name + "\nMessage: " + request.params.message;
    }
    Mailgun.sendEmail({
        to: "me@ricardoruizlopez.com",
        from: request.params.email,
        subject: "Contact form. Subject: " + request.params.subject,
        text: message
    }, {
        success: function(httpResponse) {
            response.success();
        },
        error: function(httpResponse) {
            response.error(httpResponse);
        }
    });
});
```

-Función getUserRating: Función que calcula el promedio delas valoraciones de un usuario y lo retorna. Notar que para las valoraciones de los hotspots no usamos esta aproximación, sino que cada cada vez que ponemos una valoracion, se recalcula y se almacena en precalculatedRate, de esa forma al mostrar el detalle de un punto WiFi no tenemos que hacer otra petición de red y todo vamás rápido.

-Función canIrateThisUser. Para que un usuario pueda valorar a otro, ese otro debería haber enviado un mensaje al usuario que valora, de esa forma evitamos valoraciones sin sentido. Notar que por este motivo los mensajes nunca son borrados, sino que son marcados como borrados con un flag. Entonces esta función devuelve un booleano que nos indica si un usuario puede valorar a otro o no.

-Función createUserRating. Esta función tiene como entrada el id del usuario que hace una valoración y el id del usuario que recibe la valoración, además de un mensaje y de un número de estrellitas (de 0 a 5). Esta función internamente también hace uso de la función canIrateThisUser. Devuelve la nueva valoración promedio y también nos dice si es una nueva valoración o la actualización de una, ya que un usuario sólo puede valorar como mucho a un usuario una vez.

-Función createHotspotRating. Función que recibe un id de usuario que crea la valoración, un id de hotspot que es valorado, un número de estrellitas (de 0 a 5) y un comentario. Devuelve un booleano indicando si es una nueva valoración o actualización. Devuelve la valoración promedio así como el número de valoraciones. Actualiza el campo precalculatedRate en la entidad Hotspot.

-Eliminar todos los comentarios de un punto WiFi cuando es eliminado (notar que en Parse no tenemos reglas de borrado en cascada o similar ya que es una base de datos NoSQL).

```
/*
 When deleting a hotspot, delete all its ratings
 */
Parse.Cloud.afterDelete("Hotspot", function(request) {
   var query = new Parse.Query("HotspotRating");
   query.equalTo("hotspotRated", request.object);
   query.find().then(function(hotspotRatings) {
      return Parse.Object.destroyAll(hotspotRatings);
   }).then(function(success) {
      console.log("Hotspot deleted and all its ratings deleted. Name: " + request.object.get("name") + ". id: " + request.object.id);
   }, function(error) {
      console.error("Error deleting related ratings " + error.code + ": " + error.message);
   });
});
```

-Enviar notificación push al destinatario cuando el mensaje es salvado en la base de datos.

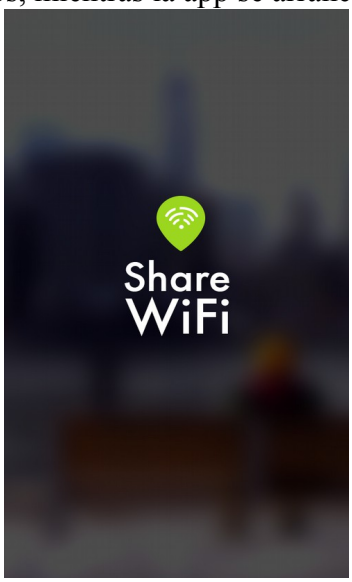
```
/*
```

```
 After a Message is saved, send a push notification to his receiver.
 */
Parse.Cloud.afterSave("Message", function(request) {

  // only do this logic when a new message is created, not when is modified (like changing unread value)
  if (request.object.existed()) return;

  // count the number of unread messages (that's the badge)
  var receiver = request.object.get("receiver");
  var query = new Parse.Query("Message");
  query.equalTo("unread", true);
  query.equalTo("receiver", receiver);
  query.count({
    success: function(number) {

      // send the push notification
      var userChannel = "uoid" + receiver.id;
      var username=request.user.get("displayName");
      Parse.Push.send({
        channels: [userChannel],
        data: {
          alert: {"loc-key": "PUSH_NM", "loc-args": [username]},
          sound: "message.caf",
          t: "m",
          badge: number
        }
      }, {
        success: function() {
          // console.log("Push notification sent successsfully after saving a message");
        },
        error: function(error) {
          console.error("Error sending push notification after saving a Message:" + error.message);
        }
      });

    },
    error: function(error) {
      console.error("Error counting the number of unread messages:" + error.message);
    }
  });
});
```

4.4. Pantallas de la aplicación.

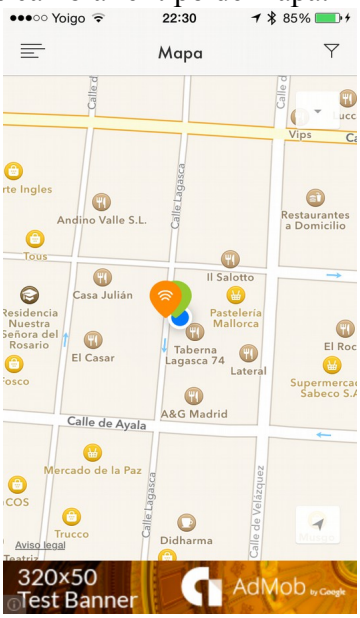
4.4.1. Pantalla de inicio.

Toda aplicación de iOS, opcionalmente aunque recomendable, tiene un archivo llamado Default.png que se muestra los primeros segundos, mientras la app se arranca.

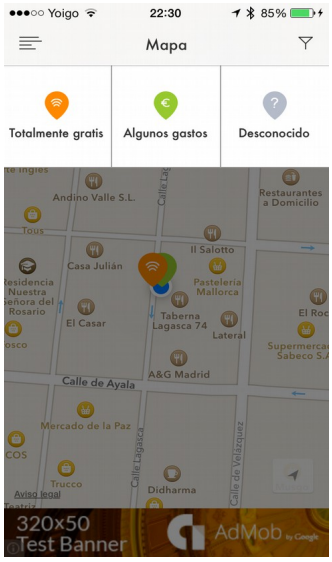


4.4.2. Ventana principal. Vista del mapa.

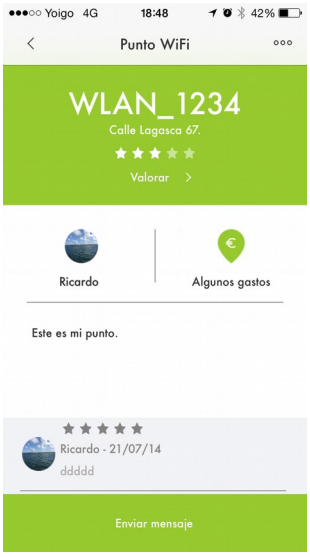
Desde aquí podemos ver todos los puntos WiFi en una determinada región del mapa. También podemos filtrarlos por tipo y además cambiar el tipo de mapa.



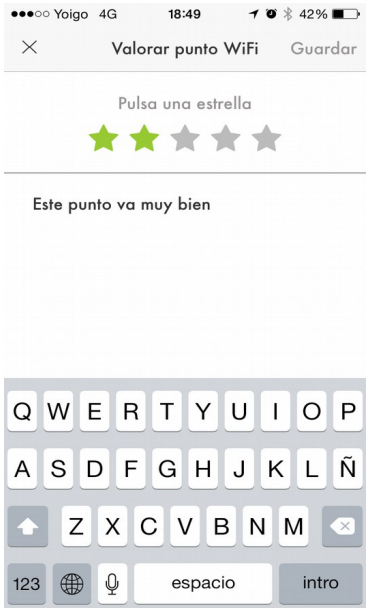
4.4.3. Vista con el filtro activado.



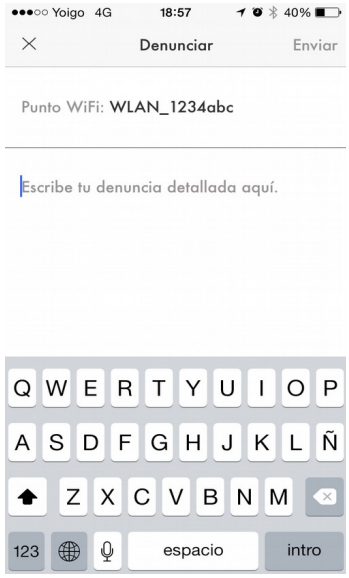
4.4.4. Detalle de punto WiFi.



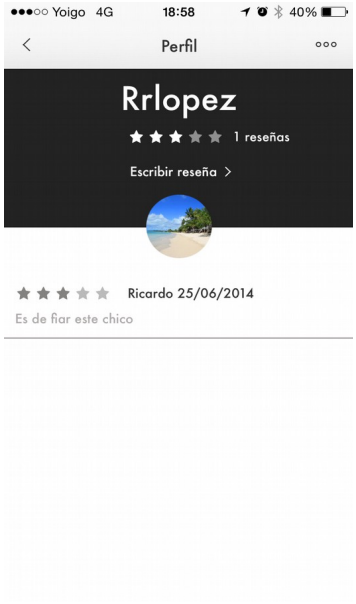
4.4.5. Valorar un punto WiFi.



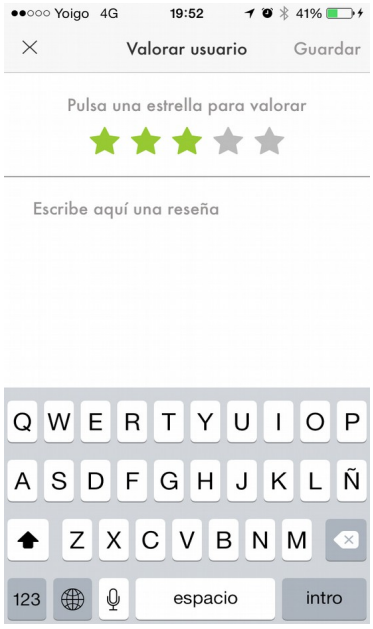
4.4.6. Denunciar punto WiFi.



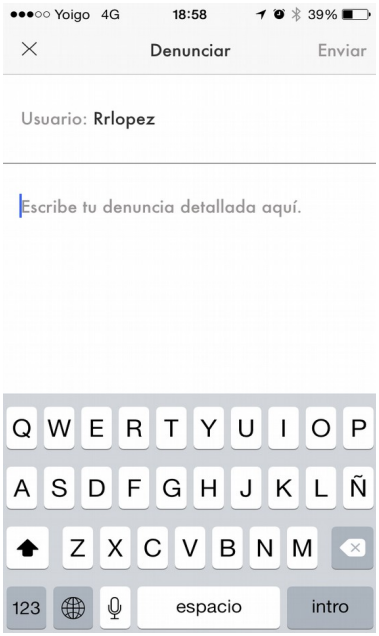
4.4.7. Perfil del dueño de un punto.



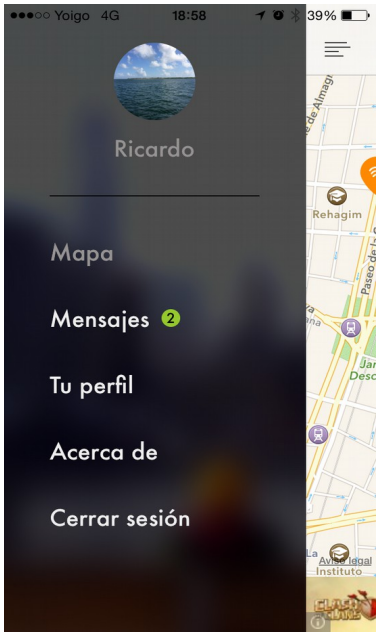
4.4.8. Valorar un usuario cualquiera.



4.4.9. Denunciar un usuario.

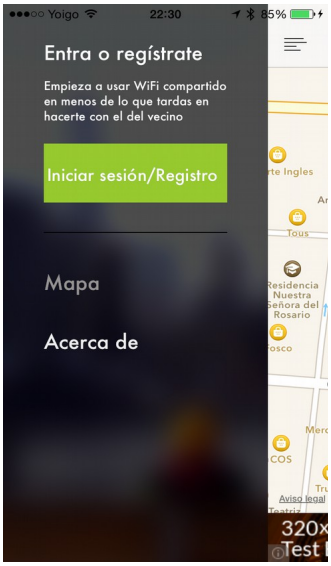


4.4.10. Menú lateral con usuario logeado.



4.4.11. Menú lateral con usuario anónimo (no logeado).

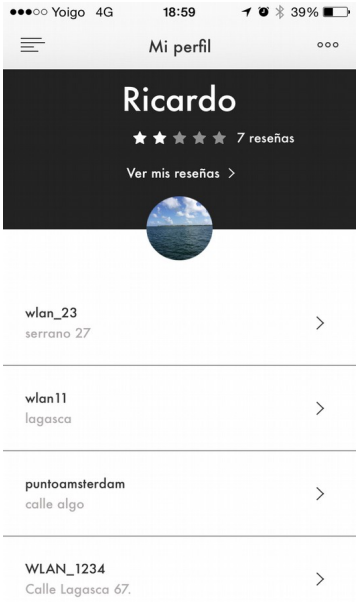
Notar que en esta vista no tenemos acceso a nuestro perfil ni bandeja de entrada de mensajes, porque no estamos logeados.



4.4.12. Inicio de sesión.

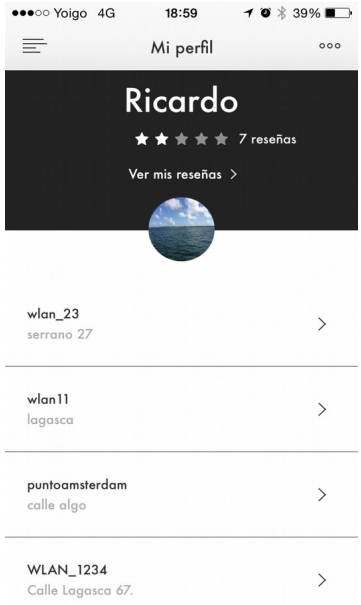
4.4.13. Registro de usuario.

4.4.14. Perfil de usuario logeado.



4.4.15. Crear punto WiFi.

4.4.16. Ver mis puntos WiFi.

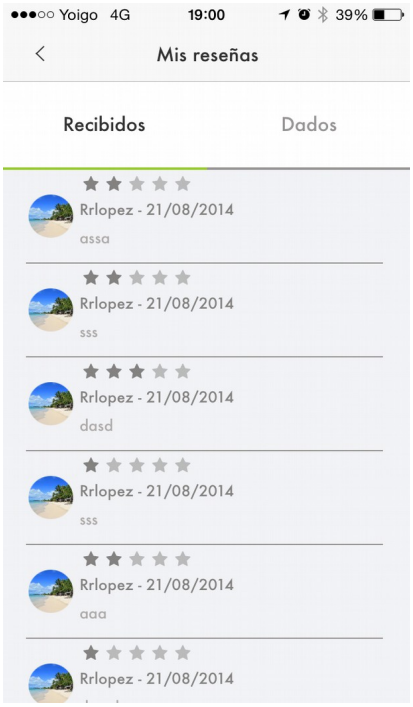


4.4.17. Ver el detalle de mi punto WiFi.

Incluyendo la posibilidad de modificarlo, excepto la posición que es inmutable.

4.4.18. Ver mis valoraciones.

Se ven tanto las las valoraciones que he dado como las que he recibido.



4.4.19. Bandeja de entrada de mensajes.

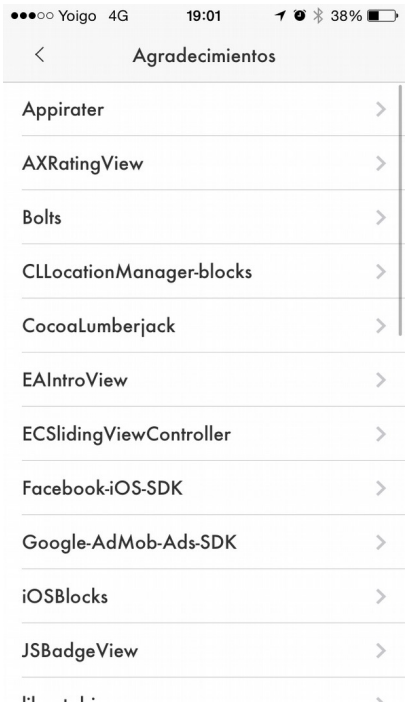
4.4.20. Detalle mensaje.

4.4.21. Escribir mensaje.

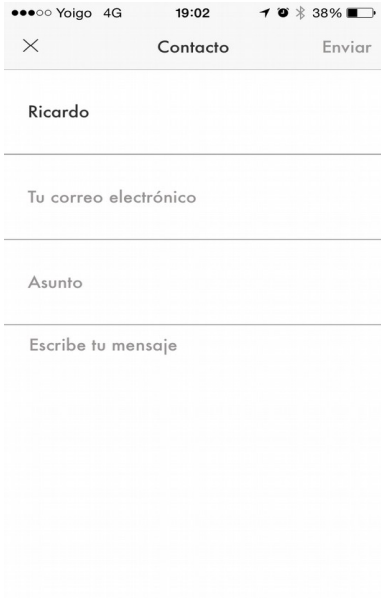
Esta ventana sale desde dos partes diferentes de la aplicación. Se puede llegar desde el detalle de un punto WiFi, pulsando enviar mensaje, o desde la bandeja de entrada, a la hora de responder un mensaje.

4.4.22. Ventana Acerca de...

4.4.23. Asuntos legales y agradecimientos.

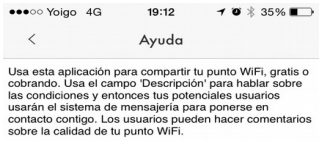


4.4.24. Formulario de contacto.



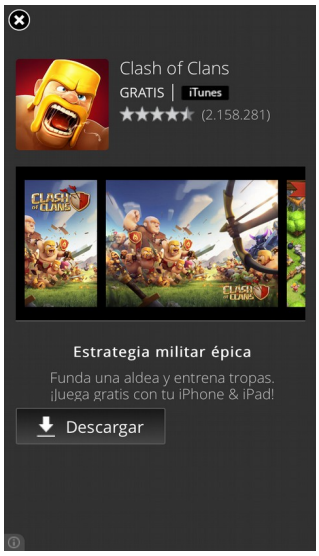
4.4.25. Ventana de ayuda.

Contiene ayuda y consejos en general.



4.4.26. Intersitial.

Anuncio a pantalla completa que se muestra después de enviar un mensaje con éxito. El anuncio es proporcionado por AdMob (Google) por lo que varía cada vez.



4.5. Diagrama de escenas.

Para la estructura de la aplicación se usará un menú lateral, que hoy en día es usado por muchas aplicaciones y los usuarios ya están acostumbrados a él. A continuación mostramos el diagrama de escenas. Por el tamaño de las hojas en una tesina no se puede ver muy bien, por eso es recomendable verlo en uno de los archivos adjuntos a esta tesina (un PDF o PNG). También se incluye el archivo original usando el software OmniGraffle.

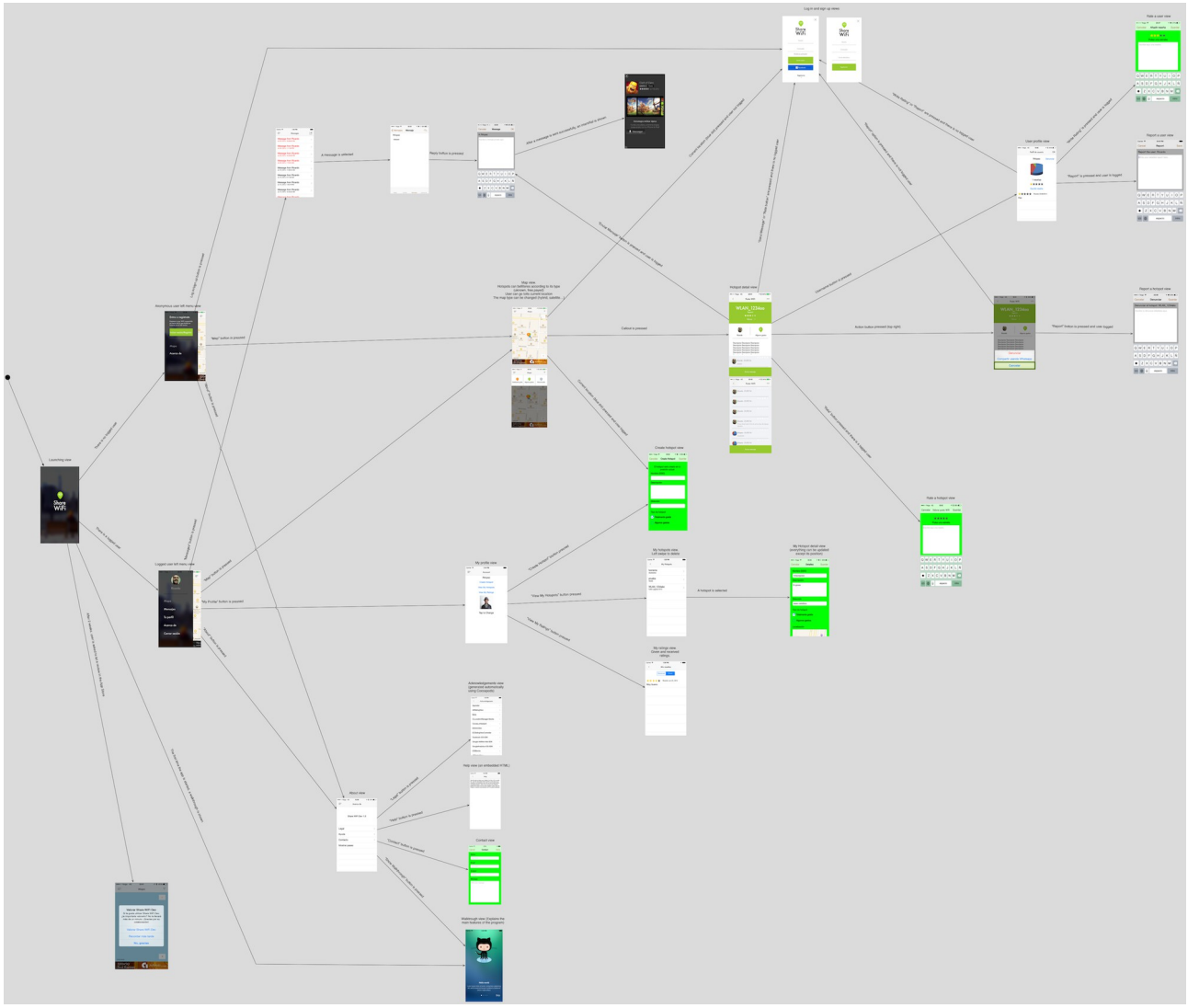


Figura 10. Diagrama de escenas de la app.

5. Implementación.

La implementación de la app se levará acabo con Xcode 5 (app nativa) usando el SDK iOS 7.0.

Además se han usado varias herramientas extra que veremos en este capítulo.

5.1. Herramientas.

- OmniGraffle ha sido usado como diagramador, para crear los modelos conceptuales de datos así como el diagrama de escenas de la app.

- Se ha usado Alcatraz como gestor de plugins para Xcode. Y se han usado dos plugins, uno para Cocoapods y otro para Injection, [20].

- Injection es un sistema que nos permite cambiar el código de una app de iPhone en tiempo real, ahorrando tiempo en el desarrollo, ya que no es necesario volver a arrancar el simulador, ni volver a la pantalla con la que estábamos trabajando, [21].

- Cocoapods. Es un gestor de librerías del que se habla en profundidad más adelante.

- Spark Inspector. Un programa muy útil que nos permite ver los interfaces gráficos en 3 dimensiones así como toda la jerarquía de vistas. Es especialmente útil para comprender qué está ocurriendo, incluso en librerías que no tenemos acceso como la de Parse. Hasta es posible cambiar los interfaces gráficos en tiempo de ejecución. Por ejemplo, uno de los componentes de Parse para iOS que facilita mucho el desarrollo se llama PFQueryTableViewController el cual automatiza la carga de una lista de entidades desde la red. Esta parte de la librería añade una vista para indicar que se está realizando una petición de red, pero esa vista no casaba con el diseño artístico.así que con Spark Inspector pude localizar qué estaba ocurriendo exactamente (ya que Parse no distribuye el código de su librería, sino simplemente los binarios).

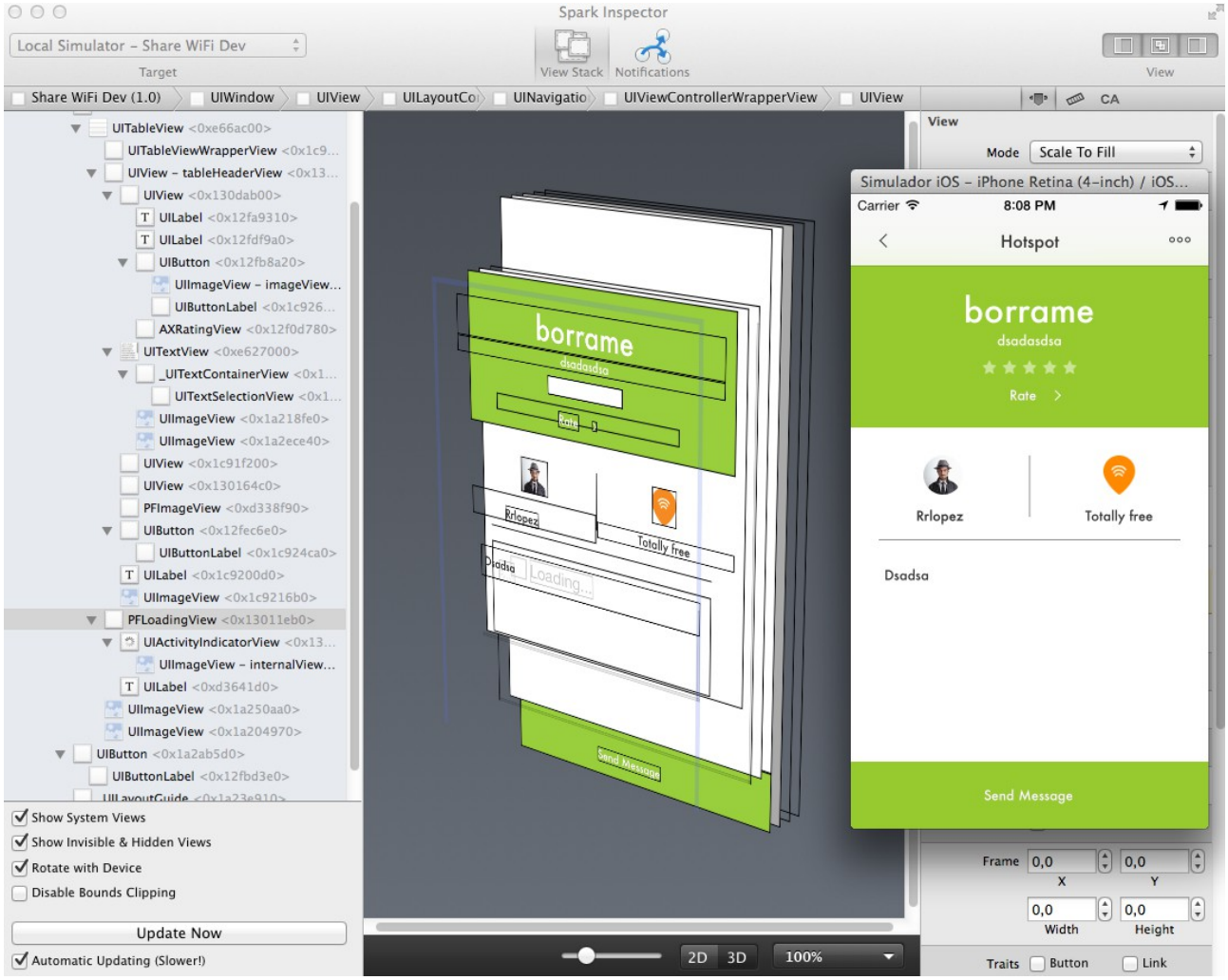


Figura 11. Ejemplo de uso de Spark Inspector en la aplicación.

En este caso descubrí una clase no documentada llamada PFLoadingView (se puede ver en la jerarquía de la izquierda de la figura de arriba), y con el siguiente código la intercepté y la quité de en medio.

```
- (void)objectsWillLoad {
    [super objectsWillLoad];
    // hack. remove or hide the PFLoadingView
    for (UIView *subview in self.view.subviews) {
        if (subview.class == NSClassFromString(@"PFLoadingView")) {
            subview.hidden=YES;
        }
    }
}
```

- Sublime Text 2. Para hacer la lógica en Parse, se usa el lenguaje Javascript. Parece que Xcode no funciona muy bien con este lenguaje por lo que decidí usar este editor de textos.

Figura 12. Ejemplo del programa creado para Parse Cloud Code usando Sublime Text 2.

- JS Format. Es un plugin para Sublime Text 2 para formatear correctamente archivos en Javascript, [22].

- Instruments. Es el profiler incluido en Xcode. Se usa principalmente para medir rendimientos y para localizar fallos en la memoria, como objetos que por alguna razón (un fallo) no son liberados correctamente.

Para la gestión del proyecto se ha usado BitBucket, que de forma gratuita permite hospedar en git proyectos privados y además incluye un gestor un tareas. Ya que es un proyecto pequeño y de un sólo desarrollador, la política en el control de versiones es subir los cambios directamente al tronco (trunk, master). Eso no es una buena práctica pero para este proyecto no importaba mucho. No he usado ningún sistema de integración continua o despliegue continuo (como Jenkins por ejemplo) debido al tamaño del proyecto. La forma de trabajar era en mi propio ordenador y probando con mi propio iPhone y iPad, ya que para probar las notificaciones push al menos hace falta dos dispositivos reales (no funcionan en el simulador proporcionado por Apple).

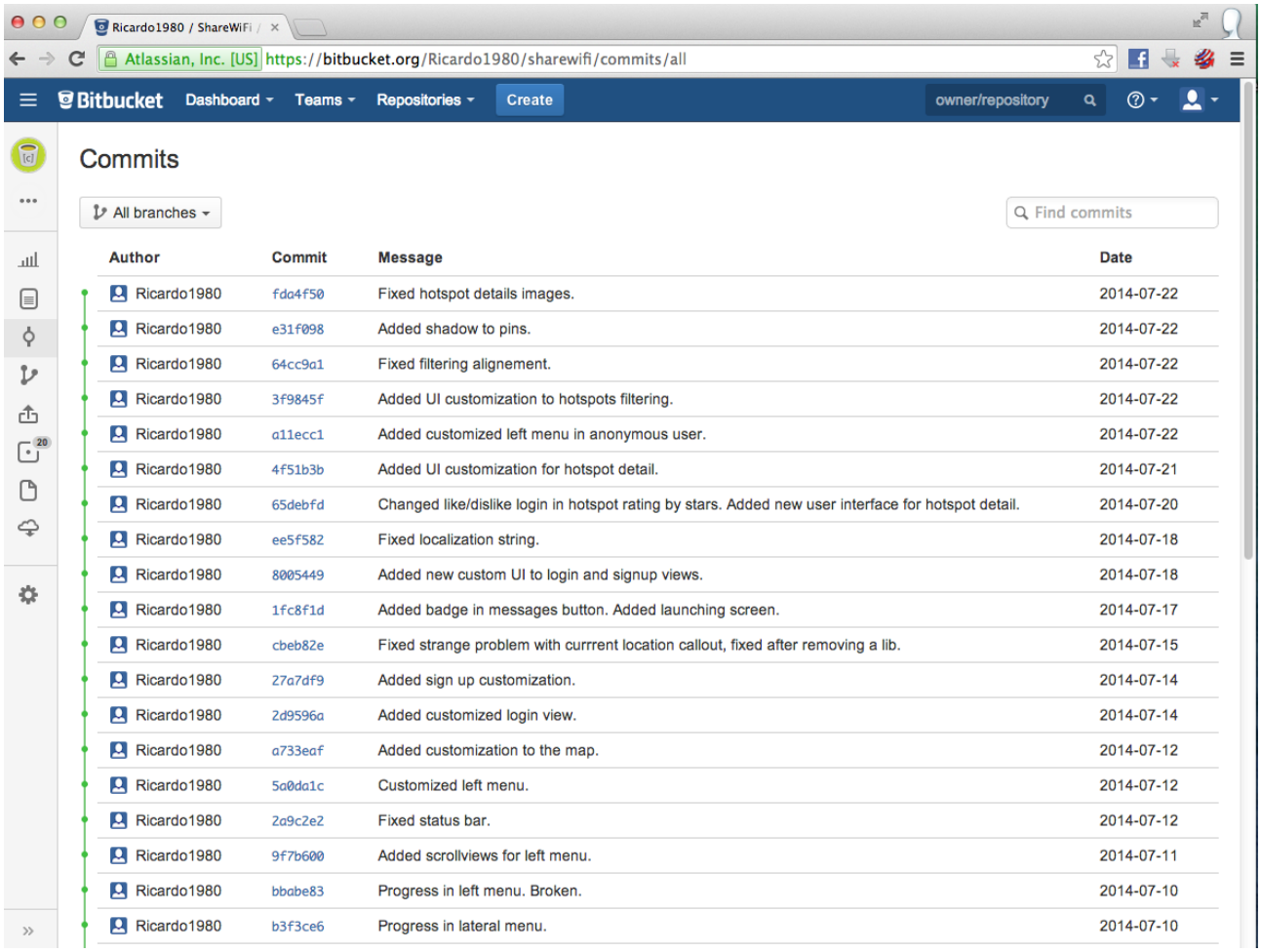


Figura 13. Ejemplo de BitBucket con la pantalla de los commits.

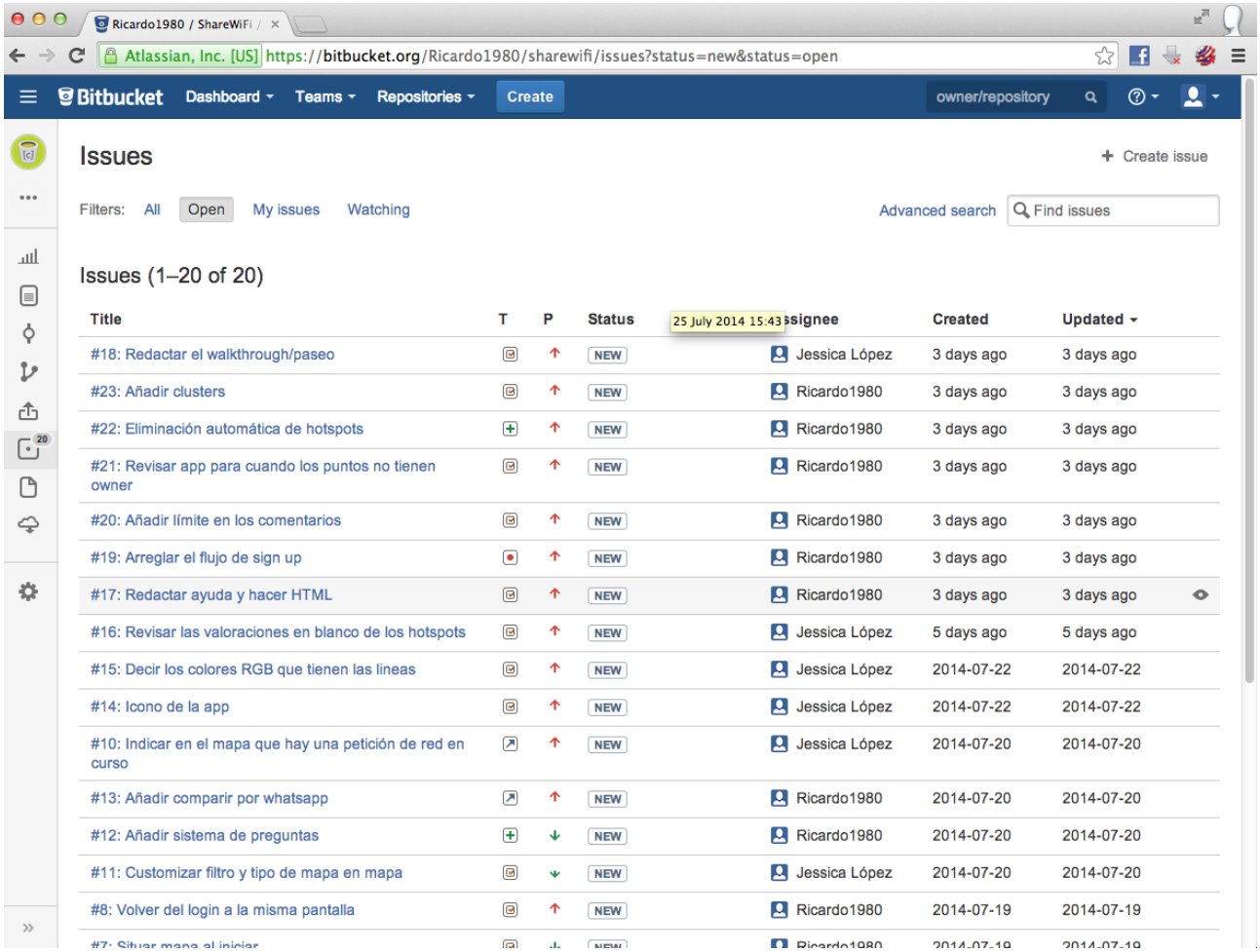


Figura 14. Ejemplo del gestor de tareas de BitBucket.

- SourceTree como cliente git, el cual se integra totalmente con BitBucket (de hecho pertenece a la misma empresa). Especialmente útil a la hora de gestionar ramas.

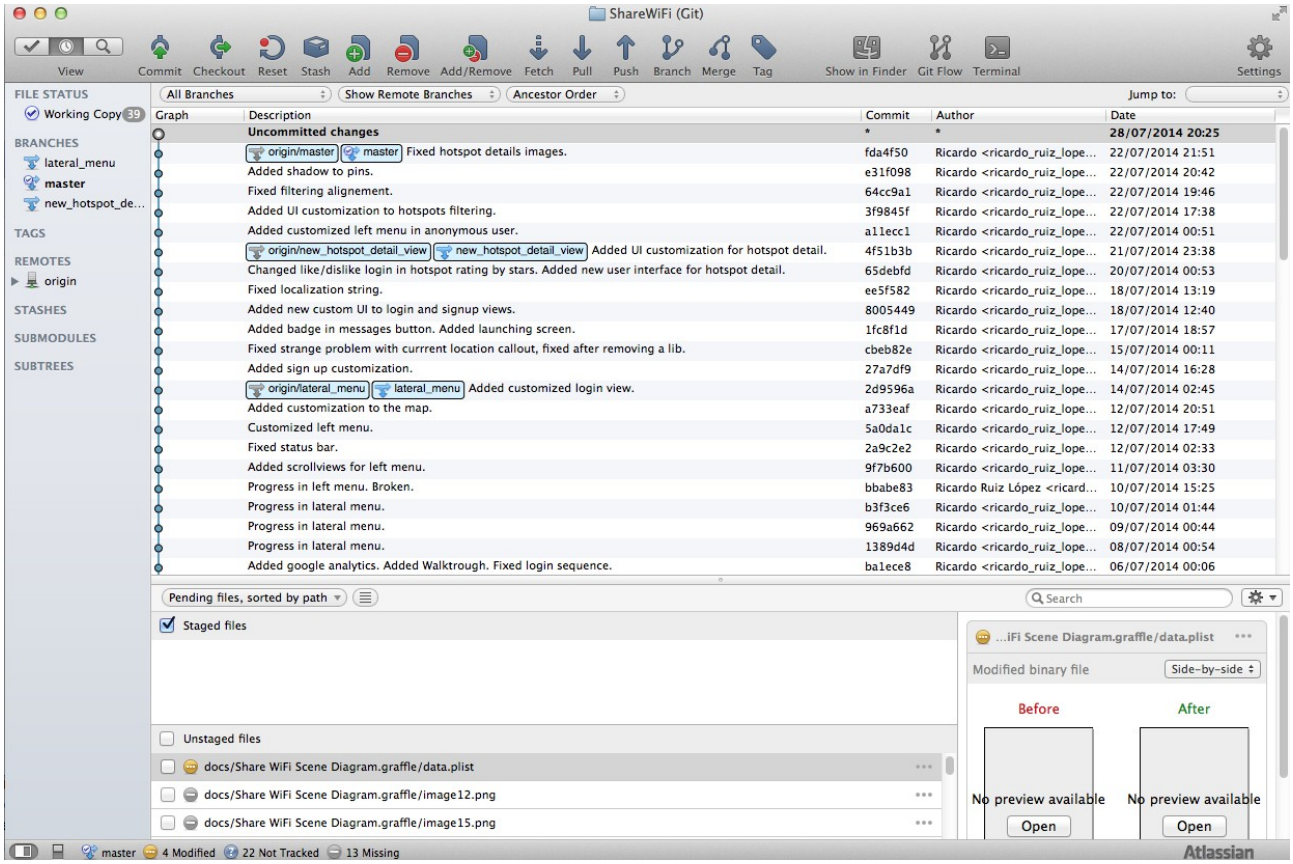


Figura 15. SourceTree con la app.

- Para el diseño final de las pantallas la diseñadora que me ha ayudado ha usado PhotoShop en general e Illustrator para los iconos.

- TestFlight para realizar despliegues a varias personas que me ayudan a probar la aplicación.

5.2. Parse

El BaaS usado en este proyecto es Parse y se usan todas las siguientes funcionalidades:

- Notificaciones push: Para avisar a otros usuarios de cuando tienen una valoración nueva (de uno de sus puntos WiFi o de ellos mismos), así como de un mensaje nuevo.

- Modelo de datos: La app no deja de ser la clásica aplicación de gestión CRUD [23] donde todo está en remoto y nada en local. Por ello es necesario una base de datos remota.

- Cloud Code [24]: Es la parte de Parse que se usa para ejecutar lógica en la parte del servidor. Parse proporciona una utilidad de línea de comandos para subir pequeños fragmentos de código escritos en javascript.

- Librería iOS [25]. Aunque todas las funcionalidades de Parse pueden usarse a través de una arquitectura REST, Parse proporciona una libraría para iOS que lo simplifica todo de manera espectacular.

La app debe ser configurada en Facebook, Parse y Apple, creando los identificadores necesarios y los provisioning profiles adecuados. Además de la configuración correcta para las notificaciones push.

Normalmente para gestionar la red en iOS, se puede el framework proporcionado por Apple, pero normalemente se usan librerías de alto nivel como AFNetworking o algunas aún más de alto nivel como RestKit que incluso se encargan de los mapeos y de la serialización y almacenamiento de datos. Sin embargo la librería iOS de Parse se encarga de toda esta complejidad, acelerando todavía más el desarrollo.

Para simular dos entornos, producción y desarrollo, he creado dos aplicaciones en Parse. Además he creado scripts y targets en Xcode para que desplegar una nueva versión del servidor sea hacer click en el ratón, en lugar de tener que usar la línea de comandos. Figura 16 y figura 17.

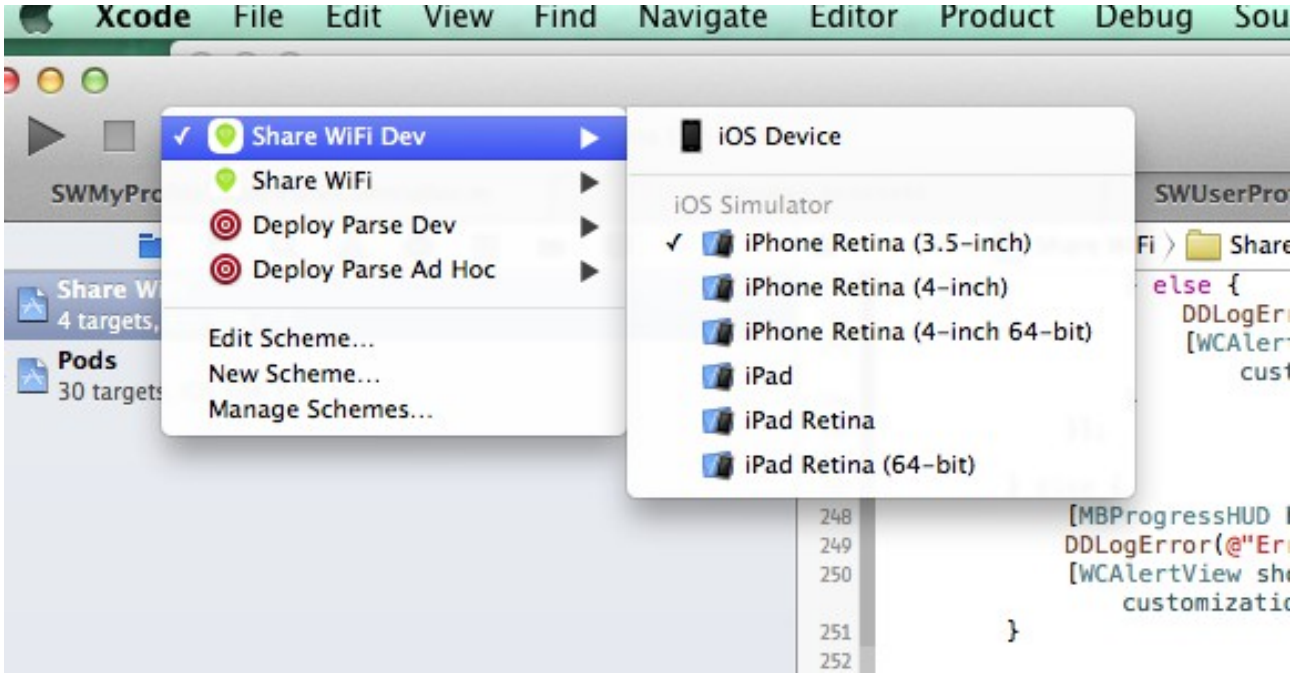


Figura 16. Ejemplo de dos entornos, tanto en la app como en los servidores.

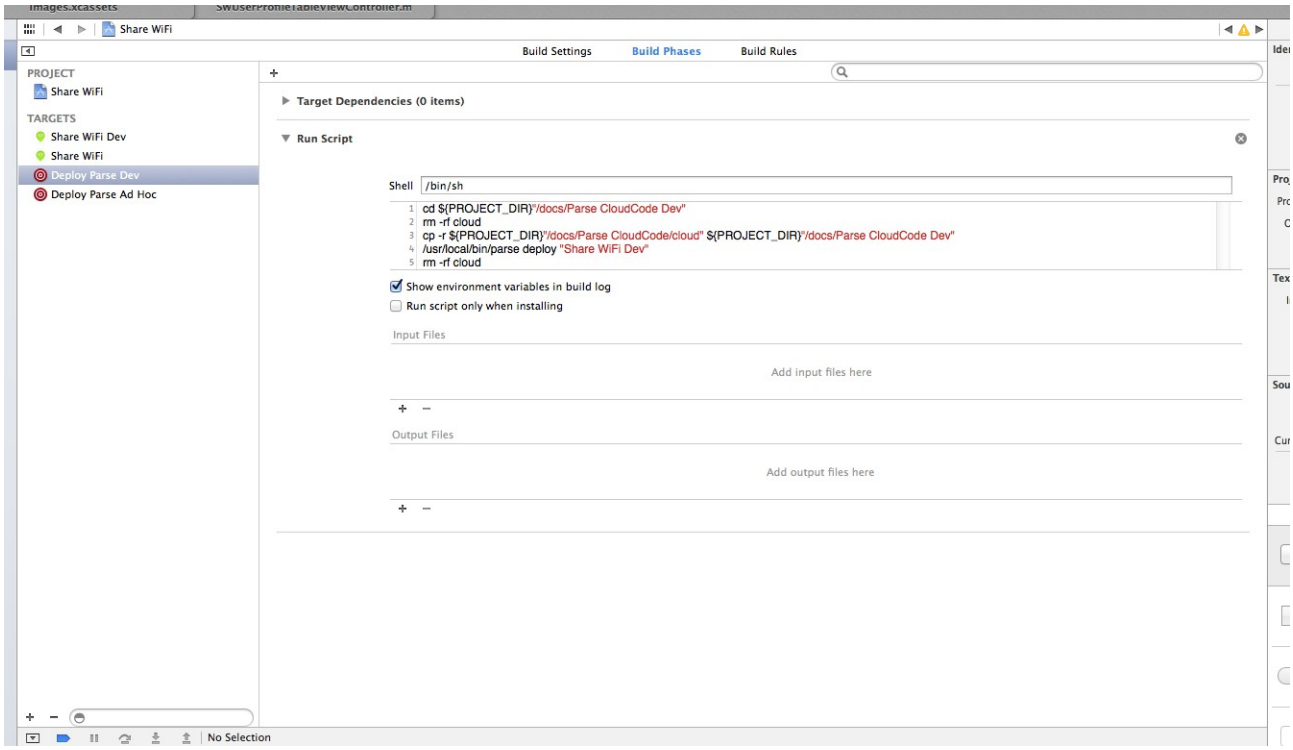


Figura 17. Script usado para subir el código Cloud Code en uno u otro entorno con un click.

Problema del versionado.

El ciclo de vida de una app (o cualquier aplicación) termina cuando se tira a la basura y ya no tienen más usuarios. Por eso, el subirla a la App Store o publicarla no significa ni mucho menos que ya que no haya que hacer cambios, de hecho todo lo contrario, ahora la app se tiene que evolucionar en función de cómo responden los usuarios.

Parte de esa evolución puede implicar la base de datos y los servicios web, o tal vez sólo sean cambios cosméticos. El caso es que si se va a cambiar los servicios web o el modelo de datos, es necesario algún tipo de versionado que nos permita evolucionar la app. Por ejemplo tal vez para la primera versión tengamos un API REST de la forma http://www.ejemplo.com/api/v1 con ciertos recursos, y luego otra versión tengamos http://www.ejemplo.com/api/v2 con otros recursos. Podríamos tener dos versiones de apps conviviendo con dos versiones de APIs, o incluso no ofrecer retrocompatibilidad y bloquear el el api v1.

Sin embargo, Parse (y la mayoría de BaaS) no tienen soporte para versionado ni para bloqueo, al menos de manera integrada o automática. Este punto puede ser un problema grave. Según Parse podemos versionar siempre que sea en forma de añadidos, es decir, añadir campos, añadir clases, añadir funciones Cloud Code... pero claramente esto no es suficiente ya que no nos permite hacer cambios importantes.

Problema de la migración de la base de datos.

En MySQL (base de datos relacional) o en Core Data (el sistema de persistencia de iOS) podemos crear planes de migración entre modelos de datos que evolucionan y mover los datos de una versión antigua a una versión moderna. En Parse, ya que no existe versionado, tampoco existe migrado de datos. Sin embargo existe el concepto de Job (una función Cloud Code) que se ejecuta a una

determinada hora. Podríamos usar esto para hacer migraciones, aunque también habría otros problemas como que sería necesario bloquear la base de datos mientras se hace y no existe mecanismo para ello. Al igual que no tener versionado es un problema, el no tener integrado una forma de migración de datos es otro problema y un punto en contra de los sistemas BaaS.

5.3. Gestión de librerías.

Es normal que las apps estén compuestas por varias librerías open source hoy en día. Se puede bajar una por una y añadir el código, con los problemas de mantenimiento que conlleva, sin embargo en iOS disponemos de Cocoapods que es un gestor que automatiza todo esto, sólo necesitamos un fichero llamado Podfile y ejecutar una línea de comando (pod update) para añadir todas las librerías que utilizamos a nuestra app.

Aunque Cocoapods no es ningún estándar industrial, se ha convertido en un estándar de facto en el mundo iOS. Tanto que para que una librería sea considerada como buena, se da por hecho que tiene que tener soporte para Cocoapods.

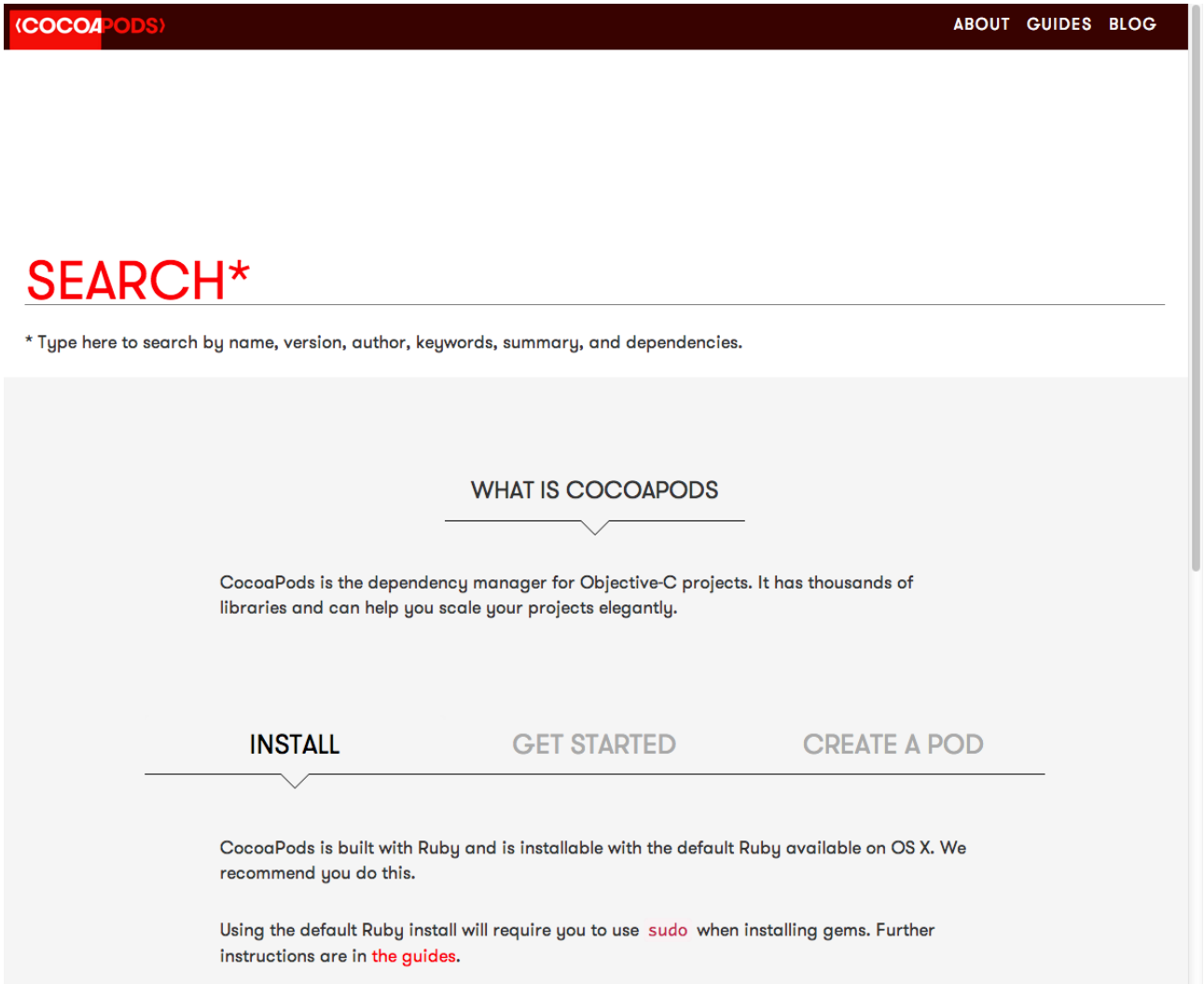


Figura 19. Página principal de Cocoapods.

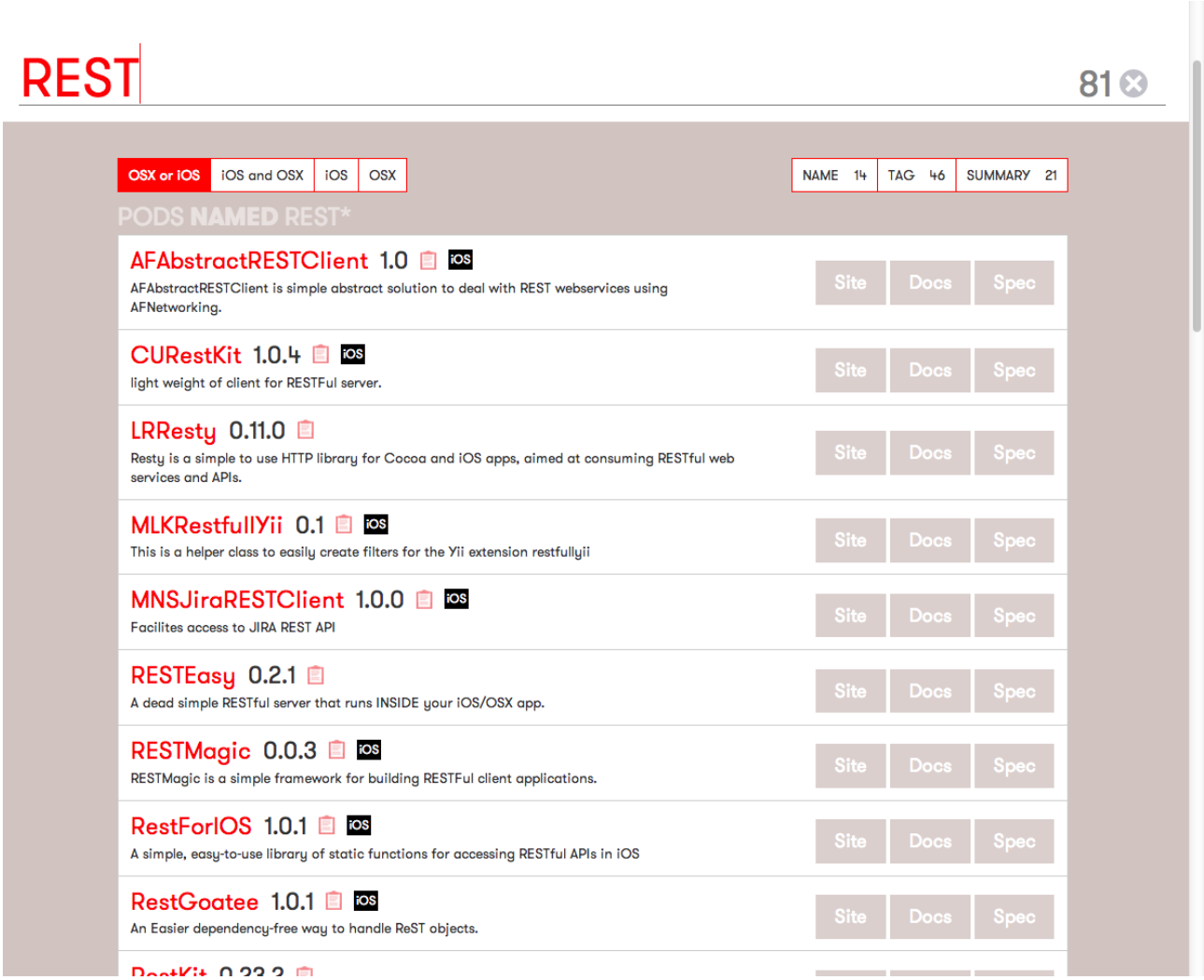


Figura 20. Ejemplo de búsqueda en la web de CocoaPods.

Una vez localizada la librería que queremos usar, debemos pulsar el icono con forma de portapapeles (entre el número de versión y el icono negro que dice iOS). Una vez hecho esto, se nos ha copiado una línea en el portapapeles de la forma:

pod 'MBProgressHUD', '~> 0.8'

Esta línea la pegamos en el archivo Podfile de nuestro proyecto y a continuación ejecutamos “pod update” (desde la línea de comandos o desde un plug in de Xcode) para que la librería se descargue. Y ya está, está instalada esa librería con esa versión en concreto.

Un recurso usado para el desarrollo de este proyecto ha sido la web CocoaControls, donde miles de desarrollladores del mundo suben sus componentes y ponen fotos de ellos. Así es fácil encontrar lo que uno quiere. Por ejemplo si en el buscador ponemos “star” podemos ver un montón de controles para poner estrellitas en nuestra aplicación. Casi todos los desarrolladores de componentes o librerías usan Cocoapods para su distribución, por lo que integrar esas librerías en nuestra app es

cuestión de segundos.

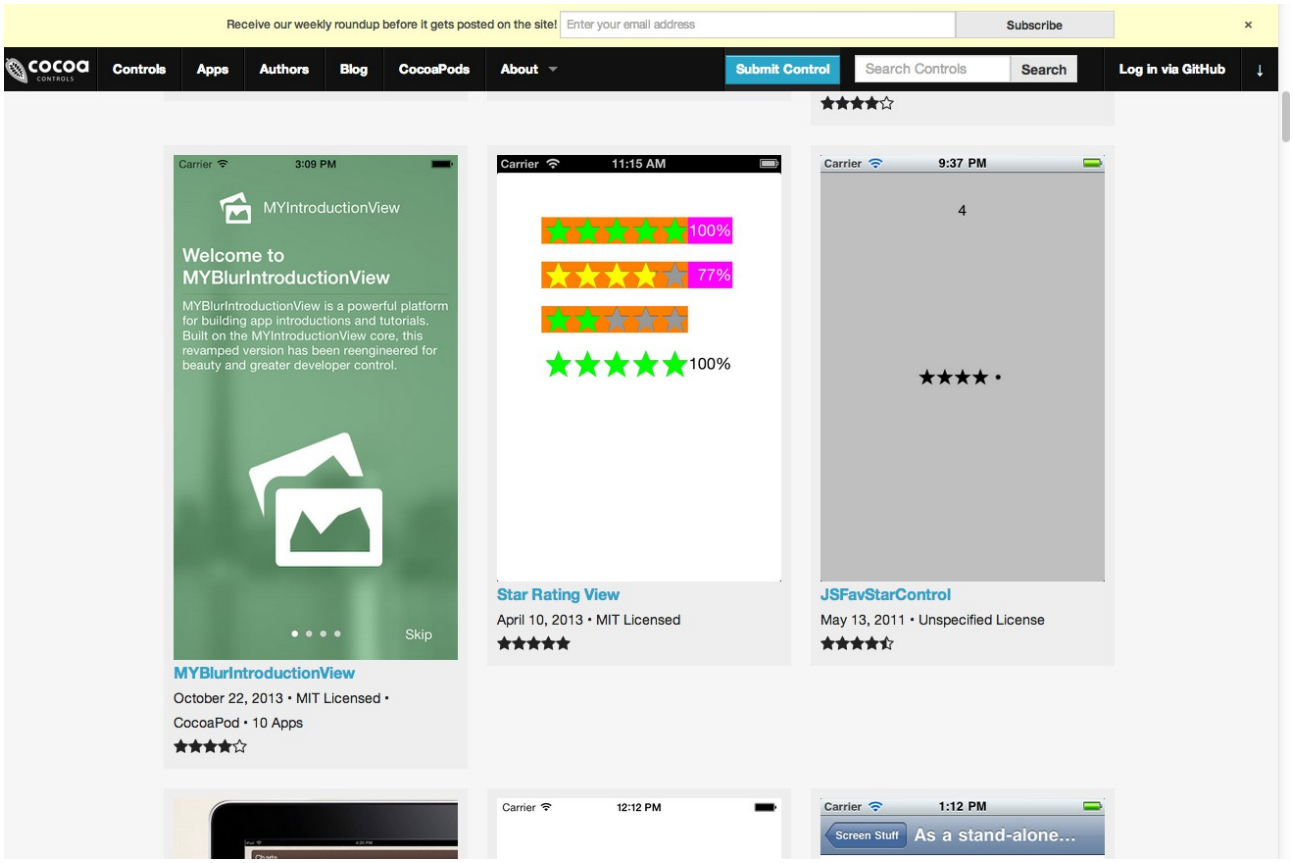


Figura 21. Ejemplo de búsqueda “star” en la web cocoacontrols.com

Contenido del archivo Podfile del proyecto:

inhibit_all_warnings! # this will disable all the warnings for all pods
platform :ios, '7.0'
pod 'MBProgressHUD', '~> 0.8'
pod 'WCAlertView', '~> 1.0.1'
pod 'VTAcknowledgementsViewController', '~> 0.11'
pod 'MPNotificationView', '~> 1.1.0'
pod 'WYPopoverController', '~> 0.2'
pod 'MKMapView-ZoomLevel', '~> 1.1'
pod 'SZTextView', '~> 1.1.0'
pod 'CLLocationManager-blocks', '~> 1.1'
pod 'UIImage-Categories', '~> 0.0.1'
pod 'iOSBlocks', '~> 1.0.2'

pod 'NSBKeyframeAnimation', '~> 0.0'

pod 'Appirater', '~> 2.0'

pod 'CocoaLumberjack', '~> 1.9'

pod 'NSString-Email', '~> 0.0.2'

pod 'SHSegueBlocks', '~> 1.2'

pod 'SparkInspector', '~> 1.2'

pod 'AXRatingView', '~> 0.9'

pod 'Google-AdMob-Ads-SDK', '~> 6.8'

pod 'TPKeyboardAvoiding', '~> 1.2'

pod 'libextobjc', '~> 0.4'

pod 'ECSlidingViewController', '~> 2.0'

pod 'JSBadgeView', '~> 1.3'

 # it must be version 19 because version 20 has a bug and email field is not shown in the sign up

pod 'Parse', '1.2.19'

Explicación de cada librería:

- MBProgressHUD. En muchas operaciones que debería bloquear la app, como ciertas peticiones de red (no todas), es necesario mostrar una barra de progreso o algo que indique al usuario que debe esperar unos segundos.

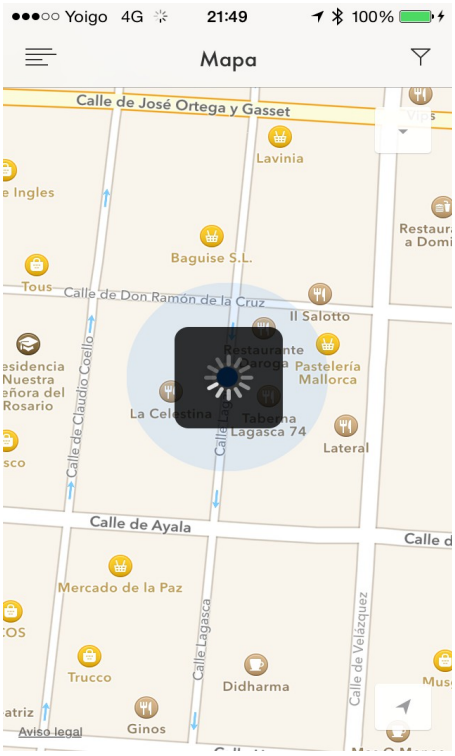


Figura 22: Ejemplo de HUD.

- WCAlertView: Para mostrar los clásicos diálogos de Aceptar/Cancelar o Sí/No se usa la clase UIAlertView, y a través de un delegado se ejecuta el código que se quiera al pulsar un botón. Esto hace que tengamos la lógica en dos partes del programa, complicando su lectura. Usando WCAlertView tenemos la lógica con un bloque en el mismo sitio donde mostramos el diálogo. Ejemplo:

```
        [WCAlertView showAlertWithTitle:NSLocalizedString(@"ALERT_VIEW_WARNING", nil)
message:NSLocalizedString(@"MY_HOTSPOT_VIEW_CONTROLLER_CANCEL_SAVE_CONFIRMATION", nil)
customizationBlock:nil completionBlock:^(NSUInteger buttonIndex, WCAlertView *alertView) {
            if (buttonIndex==1) {
                [self save];
            } else {
                [self dismissViewControllerAnimated:YES completion:nil];
            }
        } cancelButtonTitle:NSLocalizedString(@"ALERT_VIEW_NO", nil)
otherButtonTitles:NSLocalizedString(@"ALERT_VIEW_YES", nil),nil];
```

- VTAcknowledgementsViewController: Un requisito para que una librería sea compatible es que debe tener un mensaje legal o licencia. Con esta librería, podemos ver un listado de todos esos mensajes legales y de forma totalmente automática tenemos una vista para mostrarlo, que además se regenera cada vez que modificamos las librerías que hemos añadido con CocoaPods.

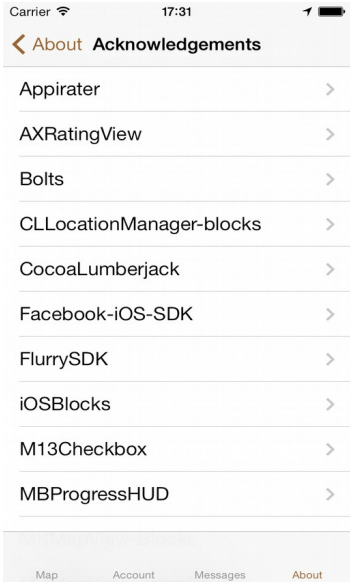


Figura 23. Mensajes legales de las librerías generado automáticamente.

- MPNotificationView: Cuando se recibe una notificación push mientras el programa está abierto, por defecto no se muestra nada, sino que por programación debería mostrarse algo. Con esta librería mostramos una ventana arriba de la app con una animación y un texto.

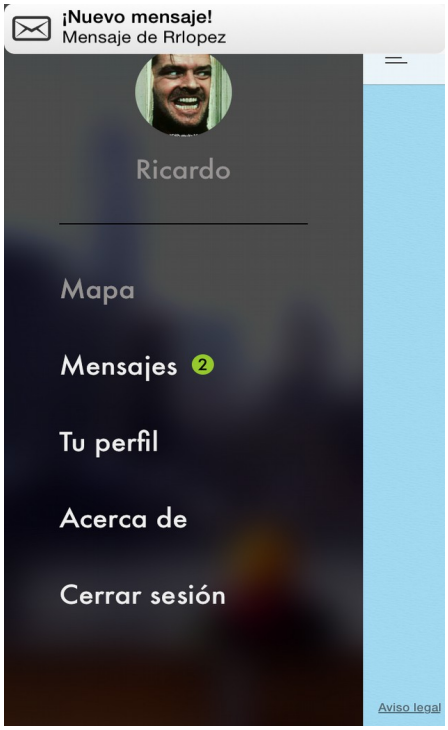


Figura 24. Ejemplo de MPNotificationView en la parte de arriba.

- WYPopoverController: En iPhone no tenemos los popover tan famosos en iPad. Esta librería suple esa carencia.

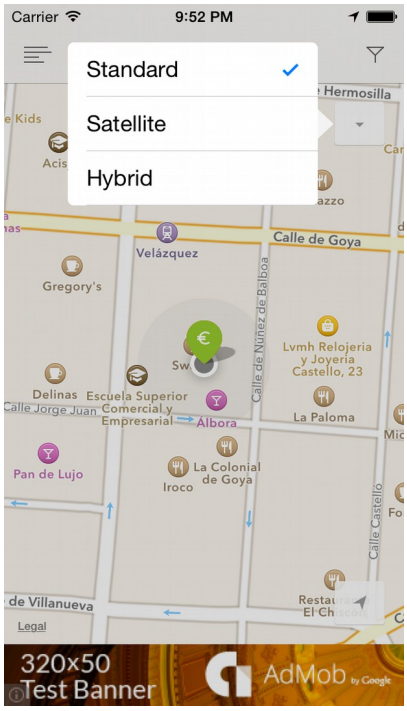


Figura 25. Ejemplo de popover en iPhone gracias a WYPopoverController.

- MKMapView-ZoomLevel: Pequeña utilidad que facilita el manejo de mapas.

- SZTextView. Pequeña utilidad que nos permite poner un placeholder el los UITextView.

- CLLocationManager-blocks: Utilidad para poder usar bloques con el API de locaclización (GPS) de Apple.

- UIImage-Categories: Usado para hacer más pequeñas la foto que el usuario puede subir en su perfil.

- iOSBlocks: Con esta clase podemos usar bloques en APIs donde Apple sólo proporciona delegados. Un uso por ejemplo fue en los UIActionSheet de la app, así tenemos la lógica en un sólo lugar y no esparcido en dos partes. A continuación ejemplo de Action Sheet usando bloques.

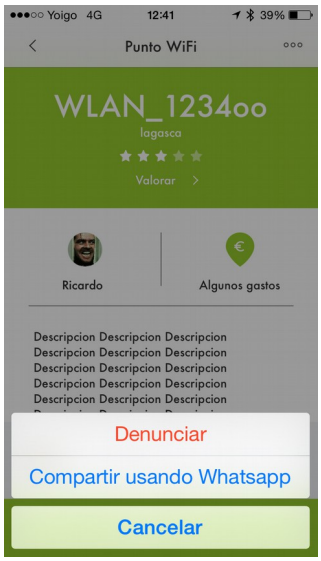


Figura 26. Ejemplo de UIActionSheet con bloques.

- NSBKeyframeAnimation: En la pantalla del mapa, los puntos WiFi cuando aparecen, se muestran con una pequeña animación de rebote. El código de la animación está en:

```
- (MKAnnotationView *)mapView:(MKMapView *)mapView viewForAnnotation:(id
<MKAnnotation>)annotation;
```

Del archivo SWMapViewController.m

```
        // super cool animation for new hotpots added to the map
        float animationDuration=randomFloat(kAnnotationViewAnimationCreationMinDuration,
kAnnotationViewAnimationCreationMaxDuration);
        NSBKeyframeAnimation *animation = [NSBKeyframeAnimation
animationWithKeyPath:@"transform.scale.x" duration:animationDuration startValue:0 endValue:1
function:NSBKeyframeAnimationFunctionEaseOutElastic];
        animation.delegate=self;
        NSBKeyframeAnimation *animation2 = [NSBKeyframeAnimation
animationWithKeyPath:@"transform.scale.y" duration:animationDuration startValue:0 endValue:1
function:NSBKeyframeAnimationFunctionEaseOutElastic];

        float maxStartingDelay=randomFloat(0, kAnnotationViewAnimationCreationMaxDelay);
        animation.beginTime = CACurrentMediaTime() + maxStartingDelay;
        animation2.beginTime = CACurrentMediaTime() + maxStartingDelay;
```

```
        [myPersonalView.layer addAnimation:animation forKey:@"transform.scale.x"];
        [myPersonalView.layer addAnimation:animation2 forKey:@"transform.scale.y"];
```

- Appirater: Es una librería para que al cabo de un par de semanas de uso de la app, pida al usuario una valoración en la App Store.

Figura 27. Ejemplo de Appirater en la app.

- CocoaLumberjack: Para tratar los errores de forma más elegante e incluso para verlos con colores en la consola de Xcode.

- NSString-Email: Clase para validar si una una cadena es un mail o no, a través de una expresión regular.

- SHSegueBlocks: Una de las librerías más sencillas de usar y a la vez más útiles que hay. Nos evita tener que usar variables de instancia para pasar argumentos en las transiciones a otras vistas (a través de los segue). Por ejemplo, cada vez que en el detalle de un punto WiFi se hace click/tap en su dueño, se abre una ventana con el perfil de ese usuario. Obviamente hay que pasar una variable que representa a ese usuario. En la app el código al pulsa la foto del dueño y que abre su perfil es:

```
- (IBAction)onHotspotOwnerNameButtonPressed {
    [self SH_performSegueWithIdentifier:@"SWUserProfileNavigationController"
andDestinationViewController:^(UIViewController *theDestinationViewController) {
        UINavigationController* nc=(UINavigationController*)theDestinationViewController;
        SWUserProfileViewController* vc=(SWUserProfileViewController*)nc.viewControllers[0];
        PFUser* hotspotOwner=_hotspot[@"owner"];
        vc.user=hotspotOwner;
    }];
}
```

En caso de no usar la librería SHSegueBlocks, tendríamos que haber puesto parte de ese código en el método:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender;
```

Y además en caso de tratarse de tablas, una variable de instancia extra. Es por ello que esta sencilla librería simplifica y hace más legible el código.

- SparkInspector: Librería para integrar la app con el programa de escritorio Spark Inspector. Para depuración de los interfaces gráficos en tiempo real.

- AXRatingView: Componente para dibujar estrellitas. Una ventaja es que puede pintar partes de entrellitas, no es necesario que sea un número entero. Es especialmente útil para los promedios de las valoraciones.

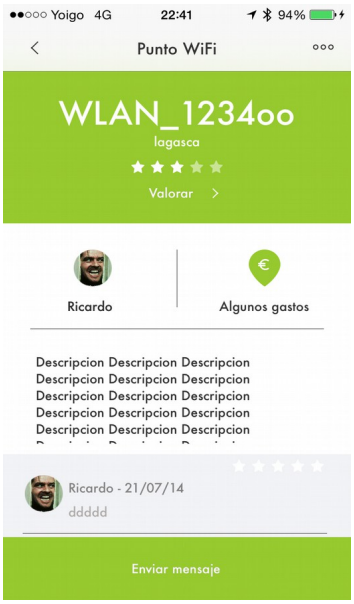


Figura 28. Uso de AXRatingView para las estrellitas.

- Google-AdMob-Ads-SDK: Librería de Google para mostrar anuncios de la red AdMob.

- TPKeyboardAvoiding: Esta librería gestiona el teclado automáticamente ante los formularios con componentes para introducir datos, ya que eleva en la pantalla dichos componentes, en caso contrario quedarían detrás dela pantalla y no podrían verse.

- Libextobjc: Esta librería resuelve de manera elegante el problema de los bloques y los ciclos con self. Si se usa self en un bloque, lo más probable es que se haya cometido un ciclo (A retiene a B y B retiene a A) de forma que el objeto jamás sería liberado. Con esta clase se resuelve al añadir los macros @weakify y @strongify. Ejemplo:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    @weakify(self);
```

```
    [[NSNotificationCenter defaultCenter] addObserverForName:kHotspotCreatedNotification
object:nil queue:nil usingBlock:^(NSNotification *note) {
        @strongify(self);
        [self.mapView.delegate mapView:self.mapView regionDidChangeAnimated:NO];
    }];
}
```

- ECSlidingViewController: Es el componente que se encarga de las vistas laterales, hoy en día tan populares en iOS.

- JSBadgeView; Es el componente que usamos en una de las listas laterales (la de usuario logeado) para mostrar el número de mensajes que quedan sin leer.

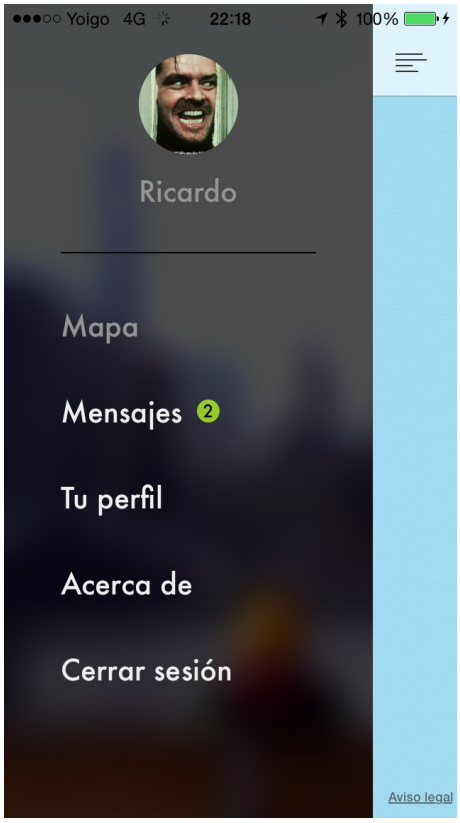


Figura 29. Ejemplo de JSBadgeView para indicar el número de mensajes no leídos.

5.4. Storyboard.

Básicamente hay 3 formas de realizar los interfaces gráficos en iOS.

La primera es hacerlo todo mediante código. Es complicado de mantener y lleva tiempo, aunque es muy flexible.

La segunda es mediante archivos NIB/XIB. Consiste en por cada vista, crear uno de estos archivos y usando el editor gráfico se modifica la vista. Es algo menos flexible pero mucho más productivo que hacer las vistas por código.

La tercera forma es usar los llamados Storyboards, que es una evolución de los NIB/XIB, de forma

que además de ver las vistas gráficamente,también vemos las relaciones entre ellas. Vemos qué vista nos lleva a otra vista. Es la forma más productiva de desarrollar apps, pero tiene el problema de que si varias personas modifican el archivo del Storyboard, entonces sea muy complicado o imposible fusionar los cambios.

En este proyecto he usado los Storyboards, ya que al ser un proyecto de una única persona, no hay problema con las fusiones (excepto si se usan mucho las ramas).

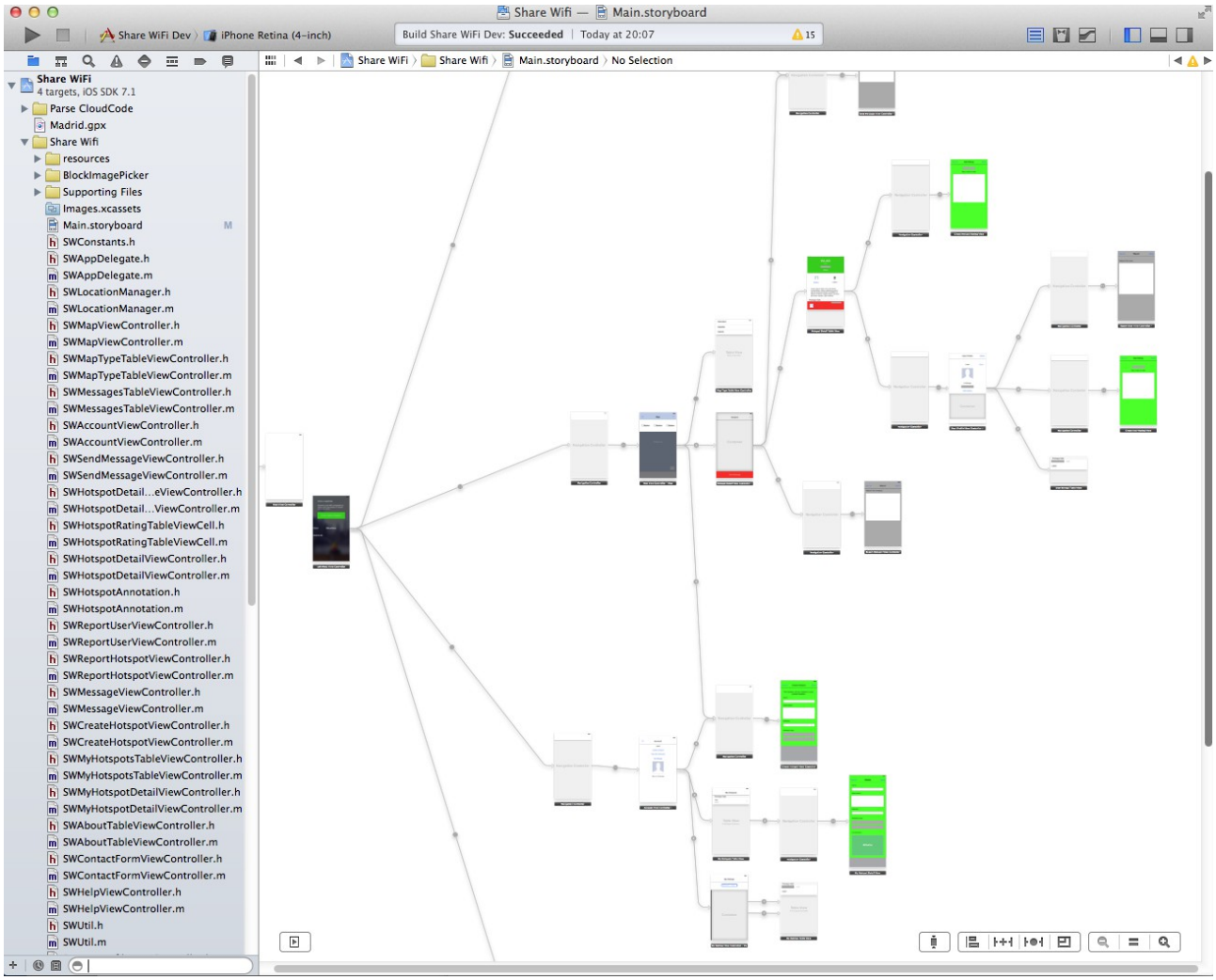


Figura 30. Xcode con el Storyboard de la app.

5.5. Autolayout.

Explicar que es una manera declarativa de posicionar los elementos de las vistas.
Para el posicionamiento de todos los elementos así como animaciones, he usado una nueva tecnología de Apple llamada Autolayouts. Consiste en lugar de decir explícitamente las posiciones y tamaños de los elementos (en puntos o píxels), lo que se hace es usar unas ecuaciones o restricciones para relacionar los elementos del interfaz gráfico con otros.
Se pueden añadir las restricciones tanto por código como gráficamente usando el editor visual de Xcode. En toda la app sólo he usado las restricciones de manera gráfica, nunca por código.
En el siguiente ejemplo vemos como la foto de perfil de usuario, tiene (entre otras) una restricción

de forma que su posición en la Y es relativa al elemento que tiene inmediatamente arriba.

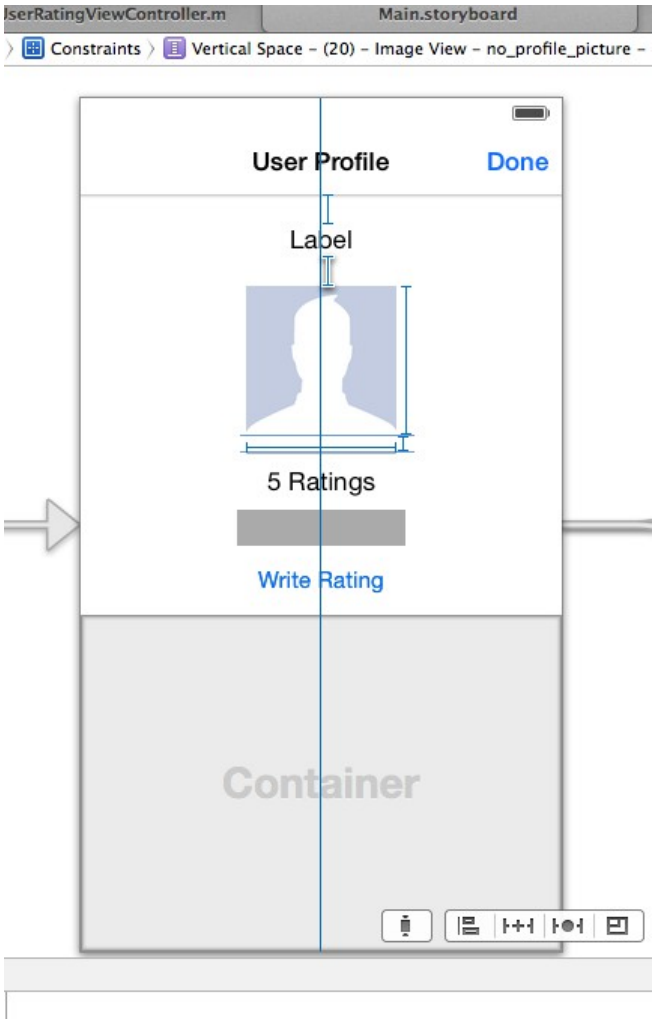


Figura 31. Ejemplo de Autolayout.

5.6. Otras cuestiones.

Una de las características que más tiempo y código ahorra usando Objective-C y los APIs de Apple, son el uso de bloques, también conocidos en otros lenguajes como closures o lambdas, [26] y [27].

Al usar bloques se evita usar el patrón delegación y la lógica de una parte del programa está en un mismo lugar en vez de repartida por varias partes, lo que facilita su lectura.

Sin embargo si no se usan adecuadamente, es posible crear ciclos de forma que los objetos nunca se liberan y con el paso del tiempo la app se degrada y el SO acaba terminándola.

Esto ocurre cuando el bloque, al capturar el contexto, también captura la variable self, creándose un ciclo de forma que el objeto que creó el bloque ya no puede liberarse.

Para solucionarlo se debe capturar una referencia débil de self dentro del bloque, como por ejemplo:

__weak ProfileViewController* mySelf=self;

Y esto mismo es lo que hacemos con las macros @weakify(self); y @strongify(self);

Ejemplo de uso en la parte del programa que se encarga de subir la foto del perfil de usuario. Notar

que incluso se ha usado un segundo bloque anidado (para crear dos peticiones de red en serie). El primero se usa para subir un archivo, y el segundo para asociar ese archivo subido con un cierto usuario en la base de datos.

```
/*
 Uploads the new picture. Updates the user entity. Updates the user interface.
 */
-(void) uploadPicture:(UIImage*)picture {

    NSData *imageData = UIImagePNGRepresentation(picture);
    PFFile *imageFile = [PFFile fileWithName:@"picture.png" data:imageData];
    UIWindow *window = [[[UIApplication sharedApplication] windows] lastObject];
    [MBProgressHUD showHUDAddedTo:window animated:YES];
    @weakify(self);
    [imageFile saveInBackgroundWithBlock:^(BOOL succeeded, NSError *error) {
        @strongify(self);
        if (succeeded) {
            PFUser* user=[PFUser currentUser];
            user[@"picture"]=imageFile;
            [user saveInBackgroundWithBlock:^(BOOL succeeded, NSError *error) {
                [MBProgressHUD hideHUDForView:window animated:YES];
                if (succeeded) {
                    DDLogInfo(@"Picture uploaded successfully");
                    [[NSNotificationCenter defaultCenter]
postNotificationName:kNewProfilePictureUploadedNotification object:self userInfo:@{@"p":picture}];
                    self.photoImageView.image = picture;
                } else {
                    DDLogError(@"Error updating user picture association. %@",error);
                    [WCAlertView showAlertWithTitle:NSLocalizedString(@"ALERT_VIEW_ERROR", nil)
message:NSLocalizedString(@"ACCOUNT_VIEW_CONTROLLER_ERROR_UPDATING_USER", nil) customizationBlock:nil
completionBlock:nil cancelButtonTitle:NSLocalizedString(@"ALERT_VIEW_OK", nil) otherButtonTitles:nil];
                }
            }];

        } else {
            [MBProgressHUD hideHUDForView:window animated:YES];
            DDLogError(@"Error uploading user picture. %@",error);
            [WCAlertView showAlertWithTitle:NSLocalizedString(@"ALERT_VIEW_ERROR", nil)
message:NSLocalizedString(@"ACCOUNT_VIEW_CONTROLLER_ERROR_UPLOADING_PICTURE", nil) customizationBlock:nil
completionBlock:nil cancelButtonTitle:NSLocalizedString(@"ALERT_VIEW_OK", nil) otherButtonTitles:nil];
        }

    } progressBlock:nil];
}
```

6. Conclusiones.

Para llevar a cabo esta app, ha sido la primera vez que he usado un BaaS. La idea se me ocurrió después de leer el libro “Application Development with Parse using iOS SDK”, [28].

Allí pude ver como la mayoría de las tareas que había que hacer en una app convencional, con Parse simplemente no era necesario. Y no sólo eso, sino que al hacerlo el mismo desarrollador de la app y no otro equipo o personas, todavía se va más rápido. Y así fue, la app fue hecha en unas pocas semanas, y muchas tareas que tenía que hacer en otras apps como todo el esqueleto para la gestión de la red, peticiones en curso, cancelaciones, multihilo, mapeado de datos... ya no eran necesarias.

Aún así, a pesar de que los BaaS aceleran extremadamente el desarrollo de una app, también tienen sus peligros, como dependencia absoluta de un vendedor en concreto (en este caso Parse/Facebok).

También hay que notar que todo el grueso sistema fue hecho en unas dos semanas, tanto servidor como la app. Sin embargo después de eso pasó a añadirse los diseños artísticos de la app que llevaron un par de meses así como la traducción a ingles que llevo casi una semana.
Por ello, aunque parezca extraño, lo que más me ha costado en tiempo y esfuerzo, ha sido personalizar el interfaz gráfico.

7. Trabajos futuros.

7.1. Promoción y márketing.

Hoy en día hay poner una app en la App Store es como poner una gota en el océano. Es necesario saber promocionarla adecuadamente y que gane exposición, de lo contrario se diluirá entre miles y miles de apps (ya se rebasó hace tiempo el millón de apps). Especialmente en las apps donde es necesario una masa crítica de usuarios como son las redes sociales o app similares, como es el caso. Por ello se debe crear un plan adecuado. Una sugerencia son los anuncios de Facebook, que tienen un alto nivel de segmentación. Por ejemplo se podría hacer campañas donde hay ciudades de universitarios, ya que es la gente más propensa a usar una app como esta (tener Internet por poco dinero y de manera flexible).

7.2. Otros smartphones.

El pastel de los smartphones en estos tiempo está repartido entre iOS y Android (ver siguiente figura). Podemos verlo en el siguiente gráfico. Es por ello que cualquier app destinada a las masas debe como mínimo tener versión iOS y versión Android. En este proyecto sólo he hecho la versión iOS porque soy experto en esa plataforma pero no en Android.
Dicho esto, tenemos otra ventaja de los BaaS, y es que podríamos bajar la librería de Parse para Android y hacer una app fácilmente. Nada del servidor necesita ser modificado, ni la lógica ni el modelos de datos. El único punto donde hay una ligera diferencia es en la forma de enviar las notificaciones push, pero eso Parse ya lo tiene en cuenta automáticamente.

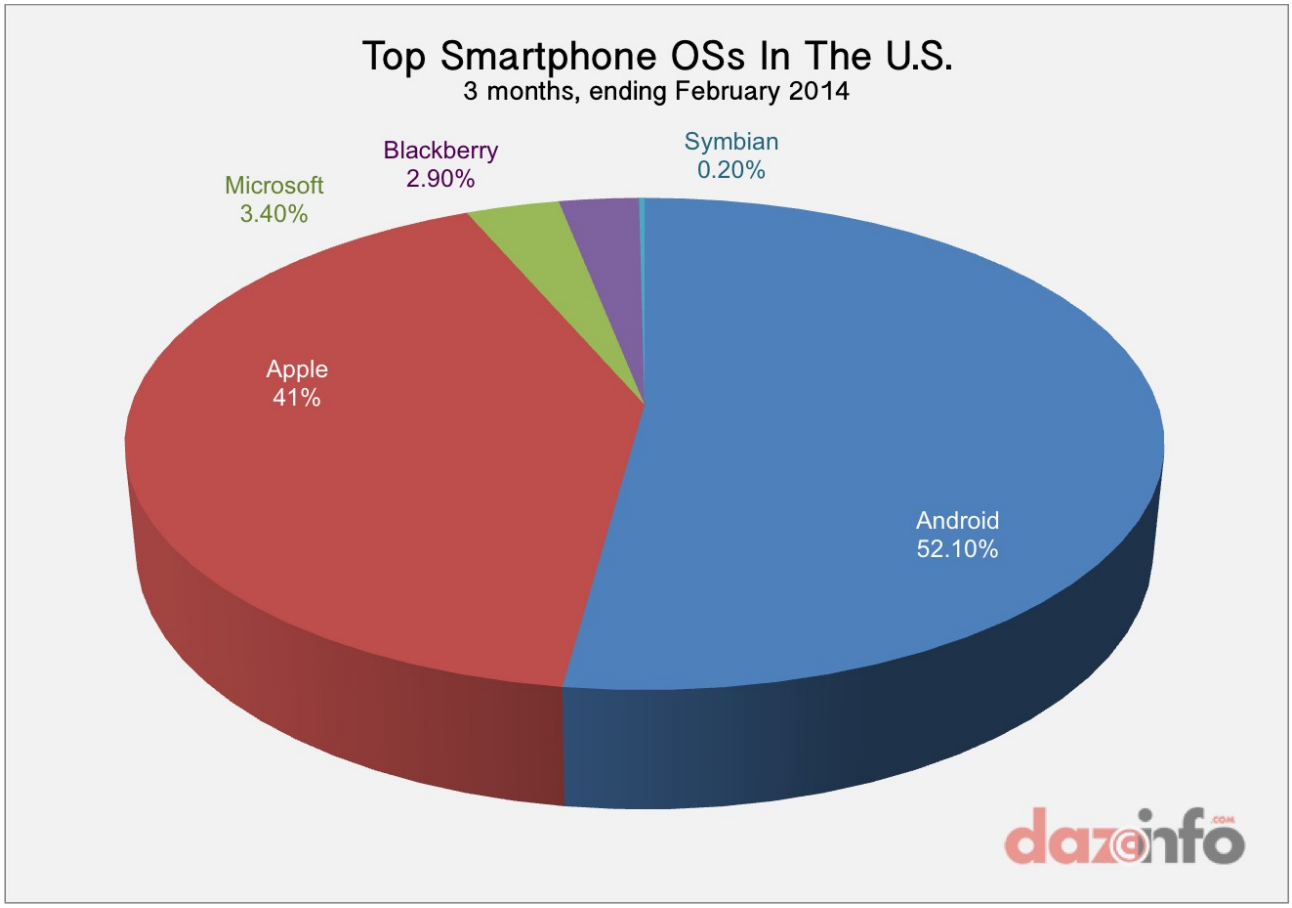


Figura 32. SO móviles en EEUU, [29].

7.3. Gestión de pagos y contraseñas.

Un par de ideas que podría mejorar drásticamente la app (y el nivel de automatización, liberando a los usuarios de tareas) es la gestión de los pagos y la gestión de las contraseñas.
Podría gestionarse los pagos mediante suscripciones de Apple o pagos periódicos de Paypal a la gente que usa tus puntos WiFi.
También podría hacerse algún mecanismo para avisar a toda la gente que te paga de que has cambiado la contraseña (porque un usuario abandona el servicio por ejemplo, o por simple seguridad). Se podrían enviar avisos mediante notificaciones push y la app mostrarte la contraseña, incluso copiarla en el portapapeles para que cómodamente la actualices en tu móvil.

8. Referencias.

[1], http://en.wikipedia.org/wiki/Mobile_data_offloading

[2], http://www.adslzone.net/2014/06/23/los-routers-de-jazztel-comparten-la-conexion-por-defecto-con-los-clientes-de-movil/

[3], http://www.comparativadeadsl.com/noticias/ono-facilita-el-acceso-a-internet-wifi-en-alicante/

[4], http://www.techinvestornews.com/Tech-News/Tech-Bloggers/framework-collaborative-economy-honeycomb

[5], http://www.asymco.com/2013/01/04/when-will-smartphones-reach-saturation/

[6], http://www.infocium.com/es/gadgets/android/u-s-smartphone-market-who-is-the-winner/

[7], http://www.20minutos.es/noticia/2181839/0/taxistas/cortan-castallena/uber/

[8], http://www.preferente.com/noticias-de-hoteles/airbnb-considerada-ilegal-por-un-juzgado-de-nueva-york-239195.html

[9], http://cnmcblog.es/2010/09/14/vecinos-que-comparten-wifi/

[10], http://www.eprivacidad.es/tribunal-supremo-direccion-ip-no-prueba-suficiente-para-imputar-delito-titular/

[11], http://es.wikipedia.org/wiki/IPhone

[12], http://en.wikipedia.org/wiki/IPhone

[13], Libro “Pro Objective-C Design Patterns for iOS”, Capítulo 1, sección “MVC as a Compound Desing Pattern”.

[14], http://en.wikipedia.org/wiki/Backend_as_a_service

[15], http://www.marketsandmarkets.com/PressReleases/baas.asp

[16], http://www.programmableweb.com/news/infographic-mobile-backend-service-ecosystem/2012/02/09

[17], http://venturebeat.com/2014/02/12/paypal-closing-down-backend-service-stackmob-months-after-buying-it/

[18], http://techcrunch.com/2013/04/25/facebook-parse/

[19], https://parse.com/docs/relations_guide

[20], http://alcatraz.io/

[21], http://injectionforxcode.com/

[22], https://github.com/jdc0589/JsFormat

[23], http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[24], https://parse.com/docs/cloud_code_guide

[25], https://parse.com/docs/ios_guide#top/iOS

[26], http://en.wikipedia.org/wiki/Blocks_(C_language_extension)

[27], http://en.wikipedia.org/wiki/Closure_(computer_programming)

[28], Libro “Application Development with Parse using iOS SDK”, http://www.packtpub.com/application-development-with-parse-using-ios-sdk/book

[29], http://www.dazeinfo.com/2014/04/16/apple-inc-aapl-claims-41-3-of-smartphone-market-but-android-enjoys-the-largest-chunk-of-the-pie/

9. Apéndice.

En el CD incluido en la tesina se incluye:

Código fuente de la app correspondiente a la fecha de entrega de la tesina (un zip del repositorio que hay en BitBucket).

Un archivo OmniGraffle que es el modelo de datos conceptual, así como su PDF.

Un archivo OmniGraffle que es el diagrama de escenas de la aplicación, así como su PDF.

Un archivo que es esta tesina, tanto ODT como DOC.