Proyecto Fin de Carrera

Faborez

App social de petición de favores instantáneos

Gorka Maiztegi Etxeberria

12 de mayo de 2014

Director:
José Miguel Blanco Arbe

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Gracias a mi director de proyecto, José Miguel Blanco Arbe, por hacer de «padre» durante todo el proyecto, no por fijarme los objetivos y los plazos, sino por obligarme a hacerlo yo mismo.

Gracias a mi padre y a mi madre, Roberto y Mertxe, por confiar ciegamente en mi trabajo en Donostia y por permitirme el lujo de estudiar la carrera con total independencia y libertad.

Gracias a todas y todos los *Txapeldunes* de este proyecto que con sus aportaciones, *feedback* e ideas han logrado que Faborez haya saltado la barrera de «aplicación para PFC» para lograr ser una aplicación real. A Goiatz, Uxue, Mikel, Unai y Alberto, gracias.

Gracias sobre todo a todos los que han luchado a mi lado por defender ideales sin interés para muchos. Primero, gracias a todos los que por casualidad he ido conociendo en la representación estudiantil: a todos los compañeros de RITSI (los cuales, sintiéndolo mucho, no podría terminar de enumerar), a Ugaitz, Jon Ander, Iker, Sandra, Jokin, Alex, Goiatz, Adrian y muchos más.

Y segundo, a los compañeros de Facultad con los que solamente con ilusión y ganas llevamos a cabo un proyecto llamado Magna SIS: Ander, Jon, Manex, Fernando, Jere y Alberto. Gracias también a toda la gente maravillosa del mundo de las Junior Empresas, que con mayor o menor afinidad, he ido conociendo.

No obstante, no es generosidad dar lo que a uno le sobra sino entregar lo que más necesita. Y es por ello que dedico un párrafo a agradecer a Joseba Egia, que habiendo participado en todo lo anterior (este PFC, la representación estudiantil y las Junior Empresas), me ha ayudado sin dudarlo cuando más lo he necesitado entregando el más preciado bien: el tiempo.

Gracias a todos aquellos que en mayor o menor medida me he topado en estos maravillosos seis años de universidad, sean estudiantes, profesores, investigadores, PAS, personal de cafetería, de limpieza o incluso «enemigos».

Gracias a ti, lector, por tomarte la molestia de rescatar e interesarte por este proyecto y leer sus páginas.

«A todos aquellos que antes que nosotros estuvieron, a todos los que habéis estado en este periodo, y a los que estáis por venir, **gracias**.» (Iñaki García García[5])

Resumen

Este Proyecto Fin de Carrera ha realizado el diseño y la implementación de la aplicación social Faborez, para la petición de favores instantáneos. El desarrollo se ha realizado en un marco de integración y colaboración directa de los usuarios en el proyecto, partiendo de un *Minimum Viable Product* inicial e integrando su *feedback* en la progresiva ampliación de las características del servicio.

En implementación se han utilizado tecnologías emergentes, todas de código abierto: MongoDB y Redis para el almacenamiento de datos, Sails.js como plataforma base para el *backend* y desarrollando como clientes una aplicación web y otra Android nativa.

Laburpena

Karrera Amaierako Proiektu honek Faborez-en, berehalako faboreak eskatzeko aplikazio sozialaren, diseinu eta inplementazioa burutu du. Garapena proiektuan erabiltzaileen integratze eta lankidetza zuzena eman den testuinguru batean gauzatu da, *Minimum Viable Product* batetik abiatuz eta beraien *feeback*-a zerbitzuaren ezaugarrien etengabeko handiagotzean integratuz.

Inplementaziorako teknologia berriak erabili dira, guztiak kode irekikoak: MongoDB eta Redis datuen biltegiratzeko, Sails.js *backend*-aren euskarri nagusia izanik eta bezero modura web aplikazioa eta Android sistema eragilerako berezko aplikazioa garatuz.

Abstract

This Final Degree Project accomplished the design and coding of Faborez, a social app for requesting instant favors. The development has been carried out with the direct integration and cooperation of the users into de project, starting with an initial *Minimum Viable Product* and integrating their feedback in a continuous feature growth.

The development has been made using recent technologies, all of them based on open source: MongoDB and Redis for data storage, Sails.js as the platform supporting the backend and developing both web and Android clients.

Índice general

Índice de figuras

Índice de cuadros

Extractos de código

Capítulo 1.

Introducción

En los últimos años se ha producido una casi completa socialización de los *smartphones*. Poca gente a día de hoy no utiliza a diario un teléfono móvil con conexión a Internet y con la posibilidad de instalar en él aplicaciones de todo tipo.

Esta popularización no se ha conseguido, no obstante, por las capacidades de los propios dispositivos, sino por su apertura al mundo de los desarrolladores y la facilidad que tienen estos de ampliar su funcionalidad con programas que los usuarios pueden instalar fácilmente.

De esta manera, al igual que ha ocurrido con muchas invenciones, no solo han cubierto las necesidades que se tenían *a priori* de inventarse estas aplicaciones, sino que han traído consigo la creación de necesidades nuevas no percibidas hasta el momento por la mayoría de las personas. A saber: la gestión de la contabilidad personal, registro de las actividades de ejercicio o incluso la publicación de fotos por el mero hecho de que sean visualizadas por un anónimo. Estas necesidades son llamadas popularmente «*first world problems*», o «problemas del primer mundo» en castellano.

En este contexto de saturación del mercado de las *app*s es donde Faborez toma forma y es creada precisamente para satisfacer, y probablemente crear, otra necesidad no existente o no percibida hasta el momento: la petición de ayuda urgente y cercana a gente no necesariamente conocida.

El trabajo realizado en este Proyecto Fin de Carrera ha tenido como objetivo la concepción, diseño e implementación de la aplicación que pretende dar solución a esta necesidad aparentemente simple.

El progresivo desarrollo del producto ha sido conducido por un grupo de usuarios selectos, denominados *txapeldun*es, en los cuales se ha delegado la tarea de tomar las decisiones sobre el diseño y las características del producto. Esta integración de personas externas ha requerido tareas de gestión exclusivas, con entrevistas guiadas por cuestionarios, comunicación constante sobre el progreso del proyecto y tecnologías para la automatización del *feedback* en caso de error.

Faborez ha sido implementado con tecnologías de código libre en auge dentro del mundo del desarrollo web. Se ha utilizado Sails.js como *framework* para el *backend*, siendo este el eje angular para el servicio. El almacenamiento de los datos se ha realizado en bases de datos NoSQL: MongoDB para la persistencia de los datos de la aplicación en general y Redis para los datos de la sesión.

Se han desarrollado dos clientes: una aplicación web y una *app* para Android. Se ha descartado el desarrollo en forma de aplicación multiplataforma debido a las limitaciones de estos entornos, tras lo cual se ha optado por el sistema operativo Android por el conocimiento previo y mayor popularidad respecto a otras plataformas.

Los servicios de Google forman una pieza primordial en la arquitectura de Faborez. Por un lado, la autenticación de los usuarios se realiza mediante Google+, descargando la tarea de gestionar credenciales. Por otro, se ha hecho uso de *Google Cloud Messaging* (GCM) para el envío de notificaciones *push* a los dispositivos móviles, para así alertarles de las peticiones de favor.

Tras analizar las distintas alternativas, se ha desplegado Faborez en el servicio *Platform–as–a–Service* (PaaS) Heroku, que con poco esfuerzo de configuración y de forma gratuita ha alojado la aplicación desde el inicio hasta el final del proyecto.

Este documento, que es la memoria del trabajo realizado por Gorka Maiztegi Etxeberria bajo la dirección del doctor José Miguel Blanco Arbe durante el curso académico 2013/2014, está estructurado de la siguiente forma:

- Los **objetivos** del proyecto, explicando el problema a resolver mediante **historias de usuario**, el **alcance** que el proyecto ha abarcado y las **exclusiones** de éste.
- Las **tecnologías** que se han utilizado para desarrollar el producto, describiendo para cada una las razones que llevaron a su elección frente a las alternativas existentes.
- Los procesos de selección, gestión y motivación de los ***txapeldun*es** del proyecto. Se abordan los métodos utilizados para la recogida del *feedback* y su integración dentro del desarrollo del proyecto.
- Se describe la **arquitectura** global de Faborez, enumerando las partes que lo componen y la interacción entre todas ella. Este apartado hace un especial énfasis en cómo se integran los servicios de Google en el funcionamiento del servicio.
- Para cada una de las **partes desarrolladas** (*backend*, y clientes web y Android), se describe su funcionamiento interno, explicando el modelo de datos propio de cada uno, la lógica de negocio que tratan y las interfaces gráficas. Todos estos elementos van acompañados de ilustraciones y diagramas que ayudan a entender su contenido.
- El siguiente capítulo relata la **gestión propia del proyecto** en las áreas principales no tratadas en el resto de la memoria: el alcance, el tiempo y los costes.
- El proyecto concluye con las **valoraciones** personales como extracto de la experiencia de este proyecto, a los cuales les acompañan algunas **lecciones aprendidas** probablemente útiles en proyectos futuros similares a este.

A lo largo de toda la memoria se repiten numerosos términos y acrónimos que podrían ser desconocidos para el lector o tener, en el contexto de este proyecto, un significado distinto. Por ello, se adjunta un **glosario** que describe la definición de estas palabras, al cual conviene acudir en caso de duda.

Además, a pesar de su condición de **apéndices** de esta memoria, se debe destacar el interés de estos documentos que ponen fin al documento:

- El apéndice A enumera los permisos que la aplicación de Android de Faborez solicita al instalarse. Su gran número sorprende a primera vista, por lo que se ha considerado necesaria una explicación detallada de todos ellos.
- En la implementación del cliente móvil se encontró un error presente en una de las librerías utilizadas, en este caso la encargada para autenticar las peticiones de red mediante OAuth 2.0, desarrollado por Google. Este error, que ya ha sido reportado, se explica en el apéndice B.
- Finalmente, con el objetivo de poder integrarlo en Faborez, se ha hecho un proceso de reflexión que ha dado como fruto el diseño conceptual de un sistema federado de karma, explicado en el apéndice C. Este sistema, implementado como un servicio web externo, ofrecería poder guardar un valor de karma para una misma persona, para ser accesible a los distintos servicios que utiliza.

Capítulo 2.

Objetivos del proyecto

Habitualmente nos surgen necesidades, muchas de gran relevancia o urgencia, a las cuales dedicamos empeño para lograr una respuesta o solución. Otras son tan pequeñas que al no poder hacer nada de inmediato simplemente las ignoramos. En cualquiera de los dos casos, uno puede recurrir a las personas que tenga geográficamente cerca para pedirles un favor. Si no hay nadie cerca, se recurre a utilizar el teléfono móvil para contactar con contactos cercanos y conocidos. Al final, la necesidad puede haber sido satisfecha o no.

No obstante, en todo momento se encuentran alrededor de cualquiera centenares de personas que no tienen por qué ser conocidas. Es probable que de entre todas ellas alguien pudiera habernos ayudado en la necesidad, pero sin embargo no existe ninguna forma fácil de dar con esta persona.

La solución por lo tanto, debe tener como función principal el hacer llegar a las personas cercanas nuestra necesidad, y dar la opción para que estas personas pueden ofrecernos su ayuda, generando finalmente este intercambio.

Este capítulo explica los objetivos del proyecto, describiendo el funcionamiento de Faborez mediante historias de usuario y después se detalla el alcance del proyecto junto con las exclusiones de éste.

2.1. Historias del usuario

A continuación se describen dos historias de usuario como forma concreta de caracterizar el proyecto. Primero, el relato desde el punto de vista de un usuario con una necesidad y que realiza la petición, y finalmente el punto de vista del usuario que atiende a la necesidad del primero.

2.1.1. Petición del favor

Fulano es estudiante de Grado en Psicología. Como estudiante de la citada titulación, no tiene gran necesidad de equipos electrónicos para su carrera, por lo que suele tomar prestada una calculadora cuando la necesita.

Hoy, 20 de mayo, tiene el examen final de Estadística, de 1º. Sin embargo, 30 minutos antes del examen se da cuenta de que no ha pedido prestada previamente la calculadora que le hará falta.

Fulano entonces ejecuta Faborez, hace click en «Necesito» y escribe «una calculadora científica». Hace click en «Enviar» y, 30 segundos más tarde, otra alerta se muestra indicando que «*Mengano* está dispuesto a dejarte una».

10 minutos más tarde *Mengano* se presenta en el lugar, calculadora en mano, para prestársela. Acuerdan devolvérsela tras acabar el examen. Una vez se marcha, *Fulano* abre la aplicación y hace click en el botón «√» junto al nombre de *Mengano*, mostrando su conformidad, y este recibe puntos en la aplicación.

2.1.2. Respuesta a la petición

Mengano estudia Grado en Arquitectura Técnica en la Escuela Universitaria Politécnica de Donostia.

Un 20 de mayo, mientras estudiaba para un examen, su móvil vibra, mostrando una alerta de Faborez. La alerta muestra que un tal *Fulano* necesita una calculadora científica para un examen, que es en menos de 20 minutos. Como tiene tiempo libre, decide dejársela. Comienza una conversación con *Fulano* a través de la aplicación, y tras un breve intercambio de mensajes, decide marchar hacia la Facultad de Psicología, a 10 minutos de su posición.

Allí se encuentra a *Fulano*, al cual le presta la calculadora, y a continuación vuele a sus estudios. Más tarde recibe una alerta en su móvil, diciendo que *Fulano* ha quedado satisfecho con el favor y que algunos puntos se suman a su perfil.

2.2. Alcance del proyecto

El alcance del proyecto se divide en dos apartados. Por un lado, se describen las características del problema resuelto y por otro lado los detalles acerca de las aplicaciones desarrolladas. Estos han sido los aspectos del problema anteriormente mencionado a los que Faborez ha dado solución:

- Los usuarios podrán realizar peticiones de favor a través de la aplicación, indicando un resumen, una descripción, una categoría y la caducidad de dicha petición.

- El resto de usuarios podrán visualizar peticiones realizadas dentro de una distancia determinada y responder a estas con un mensaje. Si las peticiones no son respondidas, desaparecerán pasado el tiempo de caducidad.
- Esta respuesta dará comienzo a un hilo de mensajes entre el autor de la petición y el de la respuesta, en el que eventualmente el autor de la petición podrá indicar como satisfactoria la ayuda prestada o borrar el hilo en caso contrario.
- Un usuario podrá visualizar tanto su historial de peticiones como el de otros usuarios con los que se encuentre, pudiendo ver el estado de estas peticiones.
- Los usuarios podrán reportar peticiones que consideren como uso abusivo o que estuvieran fuera de lugar, guardándose este reporte en el servidor para su posterior gestión.

El desarrollo de las aplicaciones creadas se ha realizado en base a los siguientes puntos:

1. Existirán dos clientes: una aplicación web y un cliente nativo para dispositivos Android. Ambos clientes estarán accesibles de forma pública a través del dominio de Faborez[1] y de Play Store, respectivamente.
2. La aplicación web podrá visualizarse y utilizarse en cualquier navegador moderno compatible con las funcionalidades recogidas dentro del estándar HTML5, incluyendo los navegadores de los teléfonos móviles.
3. La aplicación Android podrá ejecutarse a partir de la versión 2.3.3 de este sistema operativo, que a fecha de finalización de esta memoria los dispositivos con esta versión suponen el 99 % del total [7]. Será necesario también que los usuarios tengan activadas la conexión a Internet y la localización en todo momento para el correcto funcionamiento.
4. Para este cliente, además, existirá un panel de ajustes que permitirá: cambiar el idioma de la interfaz entre euskara y castellano, cambiar los ajustes relativos a las notificaciones y establecer la caducidad predeterminada de las peticiones a realizar.
5. El cliente Android mostrará notificaciones cuando se produzca una petición en la cercanía, se respondan peticiones propias o haya nuevos mensajes en un hilo de conversación abierto.
6. Las implementaciones se realizarán dando prioridad a tecnologías ya conocidas y de las que se dispone de una experiencia previa positiva. De la misma forma, no se incurrirá en costes significativos durante la contratación de servicios.
7. La autenticación de los usuarios se delega al servicio Google Plus, del cual se hará uso en ambos clientes. Igualmente, primando la homogeneidad de la tecnología utilizada, se utilizarán en la medida de lo posible tecnologías de este proveedor.

[1] `http://faborez.net/`

2.3. Exclusiones del proyecto

Se excluyen del alcance del proyecto los siguientes puntos:

1. La petición de favores, generalmente relativos a servicios, independientes de la localización donde se realicen. Existen en Internet otros portales con este objetivo y que no tienen en cuenta la proximidad de los usuarios.
2. El análisis de carácter legal que resultaría necesario en una aplicación final. La aplicación realiza una importante recogida de datos personales (mensajes enviados y geolocalización) que se almacenan en el servidor. Esta recogida requiere de una consideración en las distintas consecuencias legales, sobre todo los relativos a la Ley Orgánica de Protección de Datos de Carácter Personal (LOPD) y la Ley de Servicios en la Sociedad de la Información (LSSI), lo cual sería materia suficiente para un Proyecto Fin de Carrera separado.
3. La labor de documentación y sistematización de la gestión del servidor de la aplicación. En el proyecto se pondrán en línea los sistemas esenciales para el funcionamiento de Faborez, pero no se profundizará en el análisis del rendimiento, disponibilidad o seguridad a nivel de sistema operativo.
4. De la misma forma, se excluye la realización de un análisis exhaustivo de los costes y de las características de un servidor de pago, ya que se opta por una alternativa gratuita.
5. El desarrollo de clientes distintos a web y Android.
6. La adaptación de los clientes para su uso por personas con alguna discapacidad, como podría ser la petición de favores con el uso de la voz.
7. La implementación de un módulo propio para la autenticación de los usuarios, que se ha delegado en un servicio externo.
8. No se implementará un sistema para gestionar los reportes de uso indebido que los usuarios envíen. Los reportes, además, se limitarán a las peticiones y no podrán realizarse para usuarios o mensajes.

Capítulo 3.

Tecnología a desarrollar

Un proyecto de Ingeniería Informática como este, se ha definido en gran medida por las distintas adquisiciones tecnológicas que se han ido incorporando al proyecto durante el desarrollo. Para cada una de ellas se explica la necesidad que motivó su adquisición, las alternativas existentes y las razones para la elección, y sus principales características.

Cada adquisición ha jugado un papel más o menos crucial en el proyecto: algunas de ellas forman la piedra angular del servicio y este hubiera resultado muy distinto con otra elección, como la base de datos o el *framework* del *backend*. Otros, en cambio, han jugado un papel menor y podrían ser reemplazables o incluso desechados, como la hoja de estilo.

En casi todas las tecnologías, factor un decisivo para la elección de ciertas alternativas frente a sus competidoras ha sido la experiencia previa con ellas, ya que se ha querido evitar grandes periodos de aprendizaje y los riesgos derivados.

Las secciones de este capítulo desarrollan las siguientes tecnologías: MongoDB como base de datos, Sails.js como *framework* para el *backend*, la plataforma de desarrollo móvil Android, implementación de lógica en el cliente web con Backbone.js, la hoja de estilo prefabricada Bootstrap y finalmente los servicios Heroku para el alojamiento y Bugsense para la recolección de reportes de error.

3.1. Base de datos: MongoDB

Las bases de datos relacionales se han establecido desde hace décadas como el referente de almacenamiento de datos para las aplicaciones. Los sistemas basados en *Structured Query Language* (SQL) poseen las características necesarias para usos como las transacciones bancarias: *atomicidad*, *consistencia*, *aislamiento* y *durabilidad*.

Los datos, a su vez, se almacenan en un modelo tabular que se compone de múltiples filas que a su vez contienen un número de atributos definidos. Esto da lugar a todo el *álgebra relacional*.

No obstante, el modelo relacional puede resultar no ser adecuado para algunos tipos de uso, a saber: modelos de datos en árbol o grafos, tuplas con número variable de atributos o almacenamiento de datos en clave-valor.

Las bases de datos NoSQL se han popularizado en los últimos años como alternativa a las bases de datos relacionales. Entre los motivos están la simplificación del diseño, la mejora de la escalabilidad o la búsqueda de adaptar la base datos al modelo de datos necesario.

Por otra parte, estos sistemas generalmente carecen de un lenguaje estándar como SQL y priorizan la disponibilidad sobre la consistencia.

Los principales usuarios de estas bases de datos no relacionales son aquellas aplicaciones de tiempo real o que manejan gran cantidad de datos no estructurados en tablas.

3.1.1. Elección

El proyecto no tiene, *a priori*, ninguna restricción fuerte en cuanto al sistema de almacenamiento. La complejidad del modelo de datos es pequeña y las operaciones pueden permitirse perder consistencia en favor de una mayor rendimiento.

Se escoge una base de datos NoSQL para el almacenamiento de datos. Más concretamente, se hará uso de MongoDB, tecnología popular y de *software* libre, y que además es la opción de refercnia en otros proyectos basados en la geolocalización [31].

A falta de comparativas exhaustivas de rendimiento en la red, se ha seguido el criterio de la popularidad y del tamaño de la comunidad de desarrolladores [3]. Tal y como su web indica [21], es la base de datos no relacional más popular, y la plataforma que se utilizará para el servidor lo soporta mediante numerosas extensiones disponibles[1].

3.2. Backend: Sails.js

Faborez se va a implementar como un servicio web, con clientes de varias plataformas, donde la gran mayoría de la lógica se encuentra en el servidor. Este debe soportar las siguientes características:

- Compatibilidad con MongoDB, utilizado en el proyecto como anteriormente se menciona.
- Facilidad para implementar *Application Programming Interface*s (APIs) *Relational State Transfer* (REST) que den como resultado objetos *JavaScript Object Notation* (JSON).
- Soporte nativo o fácil de autenticación de clientes mediante sesiones para navegadores, y OAuth 2.0 [16] para la aplicación móvil.

[1]Paquetes de Node.js relativos a MongoDB: `https://www.npmjs.org/search?q=mongo`

- Baja latencia, que permita el envío instantáneo de notificaciones, además de poder soportar eventualmente un gran número de usuarios concurrentes.

Además, se han tenido en cuenta los siguientes aspectos deseables:

- Que la tecnología sea *libre*[2], tanto para su uso como para leer y entender el código que se está utilizando.
- La cantidad y claridad de la documentación que da la herramienta.
- La cantidad de complementos o *plugins* que se encuentran disponibles, para reutilizar código.
- La comunidad de desarrolladores de la tecnología, que facilita encontrar documentación sobre resolución de errores comunes que cometen otras personas en sitios de terceros, fuera de la documentación oficial.

3.2.1. Alternativas a la elección

A la fecha de la decisión, ya se tenía conocimiento y experiencia con otras plataformas, que han sido descartadas, que son Symfony y Play! Framework.

Symfony Está escrito en PHP, y cuenta con más de 2000 complementos [18], denominados *bundle*s. No obstante, de la experiencia personal con este *framework* se ha observado que su rigidez puede llegar a ser un problema a la hora de modificar parte del comportamiento de su base.

Play! Framework Este *framework* se encuentra escrito en el lenguaje de programación Scala [24], un lenguaje de programación funcional, de tipado fuerte y basado en la máquina virtual de Java. Play! Framework, en contra de de la anterior alternativa, indica en su web [30] que da soporte a las características deseadas para este proyecto: REST y JSON, además de indicar estar optimizado para móviles. No obstante, y de nuevo motivado por la experiencia adquirida, se descarta esta tecnología por la falta de popularidad y por lo tanto de complementos existentes y que podrán ser necesarios en el desarrollo.

[2]El término «libre» se refiere al utilizado en «libre expresión» y no «barra libre». Igualmente, en inglés se se utiliza «*free as in freedom, not as in free beer*».

3.2.2. Tecnología escogida

La pila tecnológica escogida para este proyecto es Sails.js. Este es un *framework* basado en Node.js que unifica una multitud de paquetes ya existentes, y establece unas jerarquía de carpetas y metodología de trabajo.

Estos distintos módulos forman la pila que completa el funcionamiento de Sails.js, comenzando por el motor de Node.js, hasta el propio *framework*[15]. A continuación se explican detalladamente todas estas tecnologías, comenzando desde el más bajo nivel hasta las librerías de alto nivel, tal y como se muestra en la figura 3.1.

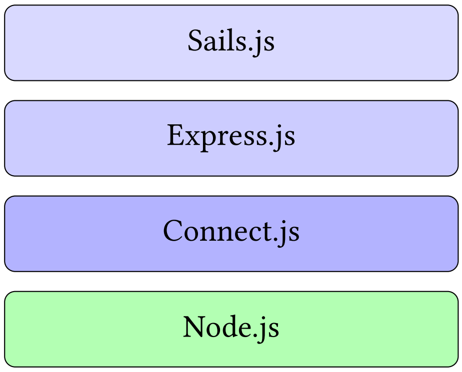

Figura 3.1.: La pila de Sails.js

Node.js

Node.js es un entorno de programación para el desarrollo de aplicaciones de alto rendimiento en el lado del servidor en lenguaje JavaScript. Su código es de licencia libre, por lo que puede ser adaptado y redistribuido.

El motor de interpretación de JavaScript de Node.js es *V8*, desarrollado por Google, que se autodefine como un motor de alto rendimiento [6].

El sistema de entrada y salida de Node.js es del tipo asíncrono y orientada a eventos[33]. Este tipo de arquitecturas están dirigidas a manejar un gran número de peticiones simultáneas y responder a estas de forma rápida, a la vez que tienen una pequeña huella de memoria y procesamiento.

De cara al desarrollador, Node.js es una capa de abstracción en JavaScript que permite crear un servidor de red en este lenguaje y realizar todo tipo de operaciones. El fin más usual es el de servidor *Hypertext Transfer Protocol* (HTTP), pero también soporta otros protocolos como WebSockets.

El extracto 3.1 es un ejemplo muy simple de servidor HTTP que responde a todas las peticiones con el texto «¡Hola, mundo!». Como se ve, el servidor es una función de JavaScript

```
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('¡Hola, Mundo!\n');
})

http.listen(1337, '127.0.0.1');
```

Extracto de código 3.1: «¡Hola, mundo!» en Node.js

que recibe la petición y escribe su respuesta en el otro parámetro. Sin embargo, puede intuirse que es necesario lidiar con las especifidades del protocolo HTTP, como las cabeceras o los códigos numéricos de respuesta, para lo cual se han creado una serie de librerías y no tener que realizar estas tareas de bajo nivel.

La comunidad ha colocado, con el paso del tiempo, como referentes a las librerías Express.js y su dependencia Connect.js[3]. Con funcionalidades distintas ambas se complementan entre ellas, tal y como se describe a continuación.

Connect.js

Connect.js es un *framework* que proporciona al desarrollador de Node.js un gran conjunto de pequeños módulos, llamados *middleware*. Estos módulos añaden funcionalidad esencial a la hora de trabajar con el protocolo HTTP, entre otros:

- Analizadores sintácticos para los datos que se envían mediante formularios y de los parámetros de la *Uniform Resource Locator* (URL).
- Utilidades para trabajar con *cookie*s, así como sesiones de los usuarios.
- Cache a la hora de servir archivos estáticos.
- Compresión de las respuestas HTTP.
- Autenticación básica mediante el sistema integrado de HTTP.
- Módulo para el registro de las peticiones recibidas.

El extracto 3.2 muestra la implementación de un simple servidor que registra todas peticiones recibidas primero, y después se hace uso del *middleware query* para extraer el parámetro «nombre» y responder con este dato.

Por lo tanto, estos *middleware*s son funciones que se ejecutan en serie, y que pueden alterar la petición, añadir elementos al resultado o incluso interrumpir la cadena, por ejemplo en

[3]Paquetes de Node.js con más *estrellas*: `https://www.npmjs.org/browse/star`

```
var http = require('http');
var connect = require('connect');

var app = connect()
  .use(connect.logger('dev'))
  .use(connect.query())
  .use(function(req, res){
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('¡Hola, '+req.nombre+'!\n');
  });

http.createServer(app).listen(3000);
```

Extracto de código 3.2: Ejemplo de registro con Connect.js

una autenticación fallida. El extracto 3.3 muestra cómo sería una función que no lleva a cabo ninguna tarea.

```
function middlewareConGranFuncionalidad(req, res, next) {
    // Llevar a cabo tareas necesarias con req (petición) y res (respuesta).
    // Una vez terminado, llamar a la función next() para dar paso al
    // siguiente middleware, en su caso.
    next();
}
```

Extracto de código 3.3: *Middleware* básico de Connect.js

Express.js

Los distintos módulos de Connect.js facilitan la implementación de la lógica de la aplicación, pero pueden quedarse cortos. Una característica común a todos los *frameworks* de desarrollo web es la posesión de un sistema de enrutamiento. El extracto 3.4 muestra cómo puede simularse esta característica con los módulos revisados hasta ahora.

Se observan los siguientes inconvenientes, entre otros:

1. Se repite código idéntico en cada una de las rutas, para comprobar si es la ruta correcta y ejecutando `next()` en caso contrario.
2. Debe existir una función final que actúe ante los errores 404.
3. Se deben establecer manualmente todas las cabeceras, incluso las del tipo de datos enviados.
4. Las rutas no discriminan según el «verbo» de HTTP utilizado.

```
var http = require('http');
var connect = require('connect');

var app = connect()
  .use(connect.logger('dev'))
  .use(connect.query());

app.use(function(req, res, next) {
  if (req.url == '/hola') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('¡Hola, '+req.nombre+'!\n');
  } else {
    next();
  }
});

app.use(function(req, res, next) {
  if (req.url == '/acerca-de') {
    res.writeHead(200, { 'Content-Type": "text/plain' });
    res.end('Ejemplo de enrutamiento con Connect.js\n');
  } else {
    next();
  }
});

app.use(function(req, res) {
  res.writeHead(404, { 'Content-Type': 'text/plain' });
  res.end('¡Error 404!\n');
});

http.createServer(app).listen(3000);
```

Extracto de código 3.4: Enrutamiento con Connect.js

Express.js es un *microframework* que solventa los problemas anteriores y añade más funcionalidad y facilidades. El extracto 3.5 muestra cómo se implementaría la anterior aplicación con el uso de este *framework*.

```
var http = require('http');
var express = require('express');

var app = express()
  .use(connect.logger('dev'))
  .use(connect.query());

app.all('*', function (req, res, next) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  next();
});

app.get('/hola/:nombre' function (req, res) {
  res.end('¡Hola, '+req.params.nombre+'!\n');
});

app.get('/acerca-de' function (req, res) {
  res.end('Ejemplo de enrutamiento con Connect.js\n');
});

app.get('/ehu', function (req, res) {
  res.redirect('http://www.ehu.es/');
});

app.get('*', function(req, res) {
  res.end('¡Error 404!\n');
});

http.createServer(app).listen(3000);
```

Extracto de código 3.5: Enrutamiento con Express.js

Además de encapsular y reducir el código de cada caso de uso, ha añadido otras funcionalidades. Por un lado, los parámetros pueden encontrarse directamente en la ruta. Por otro, se añade la función `redirect()` a la respuesta, que permite realizar una redirección sin establecer las cabeceras manualmente.

Resumiendo, Express.js implementa la lógica esencial de un servidor web sobre Node.js, pero debe dotársele a este de la estructura y funcionalidades necesarias para desarrollar una aplicación de tamaño considerable.

Sails.js

La mayoría de *frameworks* traen consigo una metodología y estructura de carpetas y ficheros ya definidas, que muchas veces es lo que los caracteriza de los demás. Express.js, en cambio, en su formato de *microframework*, no establece esta estructura y deja a la libertad del desarrollador la forma de utilización y de extensión con los módulos que fueran necesarios.

Por lo tanto, resulta necesaria otra plataforma que fije esta estructura para que la aplicación siga manteniendo su modularidad a medida que aumenta su complejidad. Por otro lado, resulta indispensable una buena integración con la base de datos, característica que no se encuentra en las anteriores librerías.

Sails.js es el *framework* elegido para este proyecto. Tiene como base Express.js y por lo tanto tiene disponibles todas funcionalidades de este. Las características añadidas son las siguientes:

- Se separa la definición de las rutas de la lógica que cada una lleva asignada. Las rutas se especifican en un archivo de configuración. La lógica de negocio se divide en una jerarquía de controladores que contienen distintas acciones. Los controladores se corresponden generalmente con entidades de datos y las acciones contienen los distintos casos de uso posibles para estos.
- La API de acceso de datos es *agnóstica* [27] respecto al sistema de gestión de bases de datos utilizado. Las entidades se definen mediante clases que describen sus atributos (nombre, tipo y restricciones) y métodos. La configuración de las conexiones a la base de datos se especifica en **ficheros de configuración** dedicados para ello.
- La transformación de los datos necesarios en JSON o *HyperText Markup Language* (HTML) es sencilla. En el primer caso basta con utilizar la función `res.json()`, dando como argumento el dato a enviar. Para HTML, se utiliza la librería de plantillas *Embedded JavaScript* (EJS), integrado en el *framework*, y que especifica un directorio concreto donde estructurar estas plantillas.
- Para cada ruta pueden integrarse **políticas de acceso personalizadas**. Estas políticas tienen el formato de *middleware* anteriormente mencionado, las cuales realizan las comprobaciones oportunas para decidir si conceder el acceso o no.
- Los archivos de configuración anteriormente mencionados son del formato JSON, por lo que no es necesario programar ninguna lógica en ellos.

3.3. Elección de la plataforma de desarrollo móvil

Una aplicación social como Faborez, basada en la inmediatez del acto de pedir un favor y de responder a estas peticiones, hace necesario que pueda ser utilizado desde el dispositivo que todo el mundo lleva encima en todo momento: el teléfono móvil. Sin embargo, el comienzo del desarrollo de una *app* móvil plantea desde el inicio un dilema, si dicha aplicación debe ser desarrollada de forma nativa para cada plataforma o si desarrollar una sola aplicación que con pequeñas modificaciones sea compatible para la gran mayoría de ellas.

3.3.1. Aplicaciones nativas

Como es bien sabido, cada plataforma o sistema operativo que ha sido creado para su uso, ha traído consigo su propio entorno de programación, herramientas y metodologías distintas. Cada uno de estos ha seguido una filosofía distinta, dependiente mucho tanto de las características de los dispositivos para los cuales se realiza el desarrollo como de la estrategia comercial de las empresas que los lanzan.

Para el caso de Android e iOS el ejemplo es claro. El desarrollo de las aplicaciones del primero se realiza mayormente en Java y se proporcionan las herramientas de compilación propias para varias plataformas de PC. En el caso del segundo, el lenguaje de programación principal es Objective-C, y el desarrollo se limita al sistema operativo de Apple, Mac OS X.

La principal ventaja de las aplicaciones nativas es su rendimiento, ya que al compilarse el código fuente a código objeto o *bytecode*, no se necesita un programa intermedio para su ejecución. También posibilitan aprovechar al máximo las funcionalidades de los dispositivos y el sistema operativo.

Sin embargo, como es obvio, se debe desarrollar una aplicación para cada una de las plataformas por separado, multiplicando el tiempo de desarrollo y sobre todo de aprendizaje. Además, se añade la complejidad de la gestión paralela de la evolución de la aplicación en los diferentes entornos.

3.3.2. Desarrollo móvil multiplataforma

La creación y popularización del estándar HTML5 ha supuesto un salto cualitativo en el mundo del desarrollo web. En el pasado el HTML no pasaba de ser un lenguaje para describir el formato de un documento con el fin de que después fuera visualizado en un navegador. La interacción más pequeña con el usuario o con el sistema operativo requería utilizar tecnologías de terceros, como Adobe Flash, con algunas desventajas: tecnologías privativas y no estándares, no disponibles para todas las plataformas y/o navegadores, de rendimiento mejorable y que requerían un proceso de aprendizaje propio.

La quinta versión de HTML trajo, en un principio, pequeños añadidos que iban encaminados a eliminar la dependencia a estas tecnologías externas en usos de Internet ya popularizados entre todos los usuarios. El ejemplo sonado, sin duda, fue la posibilidad de reproducir vídeos embebidos en la web sin necesidad de complementos.

No obstante, este germen de estandarización rápidamente se contagió al mundo del desarrollo móvil. De expandir la funcionalidad de HTML se pasó a Javascript y las APIs que esta ofrecía. Lo que en un inicio solo permitía modificar la estructura del documento, comenzó a ofrecer las interfaces para todo tipo de funciones: geolocalización, almacenamiento permanente de datos, ejecución de páginas sin necesidad de conexión. Con estos estándares, y una creciente comunidad de desarrolladores, la combinación de HTML5 y JavaScript ya era suficiente para desarrollar casi cualquier aplicación.

Webapps

En este contexto de estandarización de la tecnología, la elección del desarrollo en HTML5 + Javascript es cada vez más la opción elegida por los desarrolladores [31]. Incluso algunos sistemas operativos para dispositivos móviles han elegido directamente que todas las aplicaciones deban ser desarrollados de esta manera.

La integración de una *webapp* en un móvil es realmente sencilla: todo el código HTML y Javascript es embebido en el paquete de la aplicación, y después se crea una pequeña parte de aplicación nativa que lo único que hace es embeber una ventana de navegador y ejecutar la *webapp* en ella, de manera que esta obtiene un aspecto similar a una aplicación nativa.

Limitaciones

Con todo lo mencionado hasta este punto puede parecer que todo son ventajas del desarrollo multiplataforma respecto al nativo. Si algo está frenando su adopción para todas las aplicaciones son las limitaciones que todavía tiene en dos dimensiones: el rendimiento y la funcionalidad.

Respecto a lo primero, con HTML ocurre como con cualquier otra comparativa entre lenguajes/*frameworks* de alto y bajo nivel. Cuanto mayor es el nivel de abstracción, en este caso llegando a no depender del sistema operativo, mayor es la cantidad de intermediarios que permitan su ejecución, reduciendo el rendimiento significativamente. Este problema, con el esfuerzo que se está poniendo en la optimización gracias a la popularidad del sistema, es cada vez menor.

No obstante, el factor decisivo en la elección se encuentra en la funcionalidad ofertada, y si esta es capaz de abarcar todos los requerimientos que la aplicación en cuestión abarca. La interacción directa con el *hardware* del dispositivo, la ejecución de tareas en segundo plano, la recepción de notificaciones *push* del proveedor o la integración con características específicas de cada sistema operativo son algunas de ellas.

Tecnologías híbridas de ayuda

Existen algunas tecnologías que intentan cubrir parte de esas carencias, además de simplificar todavía más el desarrollo, creando de la parte específica de cada plataforma automáticamente, por ejemplo.

El más popular del mercado en esta categoría es Cordova [2] o su variante más conocida como Phonegap [1]. Con unos pocos comandos, realiza las labores de creación de la *app* para las plataformas que se deseen, dando como resultado archivos redistribuibles finales.

3.3.3. La elección para Faborez

En un inicio, la apuesta que se realizó para Faborez fue la de aplicación multiplataforma. Las razones son las anteriormente mencionadas, y no se veía entonces ninguna restricción. El fin era, además, abrir la aplicación para más plataformas cuanto antes, por lo que este era un punto a favor. Por otro lado, el rendimiento no pasaba de un requisito deseable.

No obstante, a medida que se desarrolló el primer prototipo, se recogió el *feedback* de los primeros *betatester*s y se completaron los casos de uso, se vio que las funcionalidades del entorno multiplataforma no eran suficientes. Las peticiones en Faborez deben ser inmediatas, notificando en un corto plazo de tiempo a los usuarios, y hacerlo a los usuarios cercanos.

Las aplicaciones web solo se encuentran activas cuando están en primer plano, por lo que no es posible que reciban notificaciones del servidor no estando en ejecución, siendo imposible tener presente siempre la localización o realizar operaciones de red en ese momento.

De la misma forma, las aplicaciones multiplataforma no realizan, para todas las plataformas, las notificaciones de la misma forma que lo hacen las nativas: se muestran unos recuadros de alerta mientras la aplicación está activa, y no mediante un icono en la barra de estado mientras no están activas, de la forma que es común.

Por estas razones, se tomó la decisión de realizar la aplicación de forma nativa y no multiplataforma, con el coste de desarrollo y aprendizaje que ello supone.

Elección de Android

Dentro de las plataformas nativas de desarrollo, la elección ha sido la de Android. Las razones que han llevado a esta decisión han sido varias, a saber:

- Android es el sistema operativo de móviles más vendido actualmente, con un 78,4 % de cuota de mercado [26]. En el caso de España, esa cifra sube al 90 % [28].
- Los dispositivos con los que se contaba para el desarrollo y prueba son todos Android, tanto los propios como la gran mayoría de los de los *betatester*s.

- La primera opción alternativa a Android sería iOS, el cual tiene un coste anual de 99 $ (71,74 €) anuales para la distribución de la aplicación. El coste de Play Store es un pago único de 25 $ (18,12 €).
- Como software libre, el código fuente de Android se encuentra en disponible en Internet y pueden recogerse ideas del mismo.

3.4. Patrón de diseño Modelo–Vista–Controlador (MVC) en el cliente: Backbone.js

Con la mejora del rendimiento de los motores JavaScript en los navegadores, las aplicaciones web han ido delegando cada vez más su lógica en el cliente y dejando tan solo una API REST a disposición de este.

Los sitios web creados mediante este sistema utilizan *Asynchronous JavaScript And XML* (AJAX) para extraer este contenido de estos, transformarlos en HTML y mostrarlos en pantalla. No obstante, el JavaScript plano tan solo cuenta con la clase XMLHttpRequest y el resto debe ser implementado por el desarrollador.

Las aplicaciones han ido además aumentando de complejidad llegando a sustituir a sus alternativas de escritorio, como en el caso de los clientes de correo y hasta suites de ofimática.

Estas aplicaciones de gran escala solo se pueden lograr de forma correcta mediante patrones de diseño, como MVC, que los organicen y aseguren su estabilidad. De esta forma, han surgido librerías de JavaScript que implementan estos modelos.

3.4.1. Backbone.js

Backbone.js [4] es una librería que estructura las aplicaciones web definiendo modelos de datos, agrupando estos en colecciones y definiendo la manera en la que se van a mostrar mediante plantillas. El ejemplo extracto 3.6 muestra un ejemplo simple de un modelo.

```
var Person = Backbone.Model.extends({
  defaults: {
    name: 'Alice',
    gender: 'female'
  }
});
```

Extracto de código 3.6: Modelo de Backbone.js

Se separan el modelo en sí de su forma de representación en HTML, su vista. En esta se define el comportamiento de la aplicación ante los eventos de la interfaz: clic en los botones, introducción de datos en formularios, etc. En el ejemplo extracto 3.7 se muestra cómo se elimina un elemento al hacer clic en su botón de eliminar.

```
var PersonView = Backbone.View.extends({
  model: Person,

  template: _.template(/* Plantilla */),

  render: function () {
    this.$el.html(this.template(this.model.attributes));
    return this;
  },

  events: {
    'click .delete-button': 'delete'
  },

  delete: function() {
    this.model.destroy();
  }
});
```

Extracto de código 3.7: Vista de Backbone.js con eventos

El otro tipo de clase principal es `Collection`, que permite agrupar varios modelos del mismo tipo para mostrar elementos similares a listas.

3.4.2. Marionette.js

Esta otra librería es un añadido a Backbone.js, que añade funcionalidad para trabajar con aplicaciones más complejas. Estas son sus características:

- Añade varias vistas complejas que permiten, por ejemplo, trabajar con colecciones: `CollectionView`, `CompositeView`, `Layout` y `Region`.
- Implementa una clase de paso de mensajes entre objetos, útil para la programación orientada a eventos.
- La clase `AppRouter` facilita enlazar las URLs de la aplicación a sus respectivos controladores.

Explicar en detalle todas sus características podría ser tema para un estudio completo, lo cual no es objeto de este proyecto ni de su memoria. Se puede encontrar más información en su web [20].

3.5. Hoja de estilo: Bootstrap

El desarrollo de cualquier aplicación web pasa por realizar un deseño de su interfaz en HTML. Esta tarea generalmente se delega, cuando es posible, en un diseñador gráfico especializado, si se quiere obtener un diseño aceptable.

La calidad de la parte gráfica de la aplicación se encuentra fuera del alcance del proyecto, y al no contar con un diseñador que realice esta tarea, se ve necesario buscar alternativas que simplifiquen esta tarea. Para ello, existen disponibles en la red hojas de estilo *Cascading Style Sheets* (CSS) prefabricadas y de licencia libre.

Estos diseños facilitan la maquetación del sitio web poniendo a la disposición del diseñador elementos comunes a la hora de diseñar: estructuras de columnas adaptables al ancho de la pantalla, formato de las tablas más atractivo que aquel por defecto de los navegadores, estilo de los textos con tipografías, diseño básico de menús de navegación, bloques de alerta y de notificación,

De entre todas ellas, son dos las más utilizadas: Bootstrap [29] y Foundation [34]. La primera está desarrollada por Twitter, que lo utiliza como base para el diseño de su propia red social. Foundation, por otra parte, es desarrollado por su empresa creadora, dedicada al diseño gráfico de portales web.

Entre estas dos alternativas, es Bootstrap con la cual se tenía mayor experiencia previa, por lo que se ha optado por esta tecnología con el único fin de reducir el tiempo de aprendizaje y desarrollo.

3.6. Proveedor PaaS: Heroku

El *backend* desarrollado para Faborez debe estar disponible a la red constantemente para que el servicio se encuentre en marcha y los distintos clientes puedan conectarse a este. Existen multitud de proveedores que ofrecen servicios de distinto tipo según las necesidades de cada proyecto.

A continuación se explican la forma en la que los distintos proveedores pueden ser categorizados según su filosofía, las distintas características de cada categoría, algunos ejemplos de proveedores y las características básicas del proveedor escogido.

3.6.1. Categorías de proveedores

Los proveedores de servicios en la nube pueden ser clasificados en tres grandes categorías, según si el servicio ofrecido es de más «bajo» o «alto» nivel. En ese orden, las categorías son *Infraestructure–as–a–Service* (IaaS), *Platform–as–a–Service* (PaaS) y *Software–as–a–Service* (SaaS).

Infraestructure–as–a–Service

Estos servicios ofrecen una infraestructura sobre la que construir la arquitectura de nuestra aplicación. El proveedor pone a disposición del cliente una instancia de un sistema operativo (normalmente virtualizado). Así, el desarrollador tiene un control total sobre esta instancia, y es su responsabilidad poner en marcha en él los distintos servicios que le hagan falta.

Un mismo proveedor suele categorizar las distintas opciones según el rendimiento de las instancias virtualizadas. Es decir, el servidor tendrá un coste mayor en cuanta más capacidad tenga este en términos de CPU, memoria RAM y tamaño del disco duro.

Uno de los proveedores más conocidos es Amazon AWS[4], que tiene disponibles instancias en centros de proceso de datos a lo ancho de todo el mundo. Otras alternativas conocidas son Joyent[5] o Rackspace[6]. El cuadro 3.1 muestra una comparación de estos servicios y su precio, con características similares y de una gama media-baja.

Proveedor	Características	Precio mensual
Amazon AWS	1 vCPU, 160 GB de disco y 1,7 GiB de RAM	31,68 $ (22,96 €)
Joyent	1 vCPU, 56 GiB de disco y 1,75 GiB de RAM	40,32 $ (29,22 €)
Rackspace	1 vCPU, 20 GiB de disco y 1 GiB de RAM	29,20 $ (21,16 €)

Cuadro 3.1.: Comparativa de servicios *Infraestructure–as–a–Service* (IaaS)

Platform–as–a–Service

Esta categoría de servicios, en vez de poner a disposición un sistema operativo completo sobre el que se tiene total control, el desarrollador tiene un servidor preinstalado capaz de ejecutar aplicaciones web de una determinada pila tecnológica.

Los clásicos hostings de Apache+PHP+MySQL sobre los que se ejecutan la mayoría de webs personales y profesionales encajan en esta categoría. No obstante, el número de plataformas que se utilizan se ha diversificado y lo mismo ha ocurrido con los proveedores [19]. Existen proveedores para Java Servlets, Ruby on Rails, o Node.js, plataforma en la que se basa Faborez.

Existen proveedores de esta categoría que tienen una tarifa gratuita, para proyectos pequeños o que están comenzando a ponerse en marcha. A partir de ahí, las tarifas escalan de forma similar a los servicios IaaS, aumentando en prestaciones, pero también añadiendo características al servidor[7].

[4] `https://aws.amazon.com/es/`

[5] `https://www.joyent.com/`

[6] `http://www.rackspace.com/es/`

[7] Addons de Heroku: `https://addons.heroku.com/`

Google App Engine, Windows Azure, Heroku, CloudBees o Cloud Foundry son algunas de las alternativas existentes en esta categoría. En el cuadro 3.2 se muestran las tecnologías con las que funcionan y si disponen de versión gratuita.

Proveedor	Tecnologías soportadas	Tarifa gratuita
Google App Engine	Python, Java, Go y PHP	Sí, con cuotas
Heroku	Java, Scala, Python, Node.js, Ruby y Clojure	Sí, una instancia
Windows Azure	.NET, PHP, Node.js y Python	Sí, con limitaciones
Cloudbees	Java	Sí
Cloud Foundry	Java, Python y Node.js	*Según proveedor*

Cuadro 3.2.: Comparativa de servicios PaaS

Software–as–a–Service

Aunque generalmente no se les suele etiquetar con este nombre, estas son las conocidas aplicaciones en la nube, contrapuestas a las aplicaciones de escritorio. Gmail, Hotmail, Dropbox o Evernote son algunos de los ejemplos más conocidos de los muchos existentes.

No obstante, estas aplicaciones en la nube quedan en el nivel del usuario y no son herramientas para desarrolladores de aplicaciones, por lo que su análisis queda fuera de este proyecto.

3.6.2. Proveedor escogido

De entre las alternativas anteriores, el proveedor seleccionado para este proyecto ha sido Heroku. Su servicio se basa en la infraestructura de Amazon AWS, y soporta Node.js entre una de sus plataformas de desarrollo, y por otro lado no tiene tiene coste económico alguno utilizando una sola instancia, crucial para este proyecto. Por otro lado, también tiene algunas otras características:

- Entre los *addon*s disponibles se encuentran dos que han sido utilizados en el proyecto: MongoDB y Redis. El primero ya se ha explicado anteriormente en el apartado 3.1, y el segundo es un motor de base de datos en memoria, que en el caso de este proyecto almacena los datos de sesión de los usuarios.

- La creación de una instancia en el servidor es trivial, pero también lo es el proceso de despliegue de nuevas versiones de la aplicación. Este se realiza mediante Git, siendo suficiente un comando para subir nuevas versiones:

```
git push heroku master
```

- Heroku permite alojar la aplicación en un dominio propio sin incurrir en gastos, característica que es de pago en los demás proveedores. No obstante, se ofrece un subdominio[8] predefinido por si no se contara con un dominio, el cual tiene activado por defecto SSL.

Las razones anteriores, añadidas a la experiencia previa ya existente con el proveedor, han llevado a escoger este servicio para alojar Faborez.

3.7. Feedback de errores automático con Bugsense

El desarrollo de aplicaciones para Android es casi idéntico al de desarrollar un programa que se vaya a ejecutar en el propio ordenador. La diferencia está en que el entorno de desarrollo compila y empaqueta la aplicación, la instala en un dispositivo Android simulado o uno real conectado por USB y lo ejecuta.

Mediante esta conexión USB es posible realizar todas las tareas de depuración que se podrían realizar en una aplicación de PC: revisar la ejecución con interrupciones, vigilar la memoria consumida, pero lo más importante es trazar los errores a las líneas de código exactas donde se producen, para proceder al arreglo.

No obstante, no siempre es posible detectar todos los potenciales focos de errores durante el desarrollo, y es probable que una aplicación se publique con errores. Los usuarios detectarían estos errores porque la aplicación se cerraría con una alerta, pero el mensaje de error no llegaría al desarrollador.

Es por ello que es necesario crear un sistema por el cual, sin depender de la voluntad o conocimiento de los usuarios, se recojan de manera automática y ordenada estos informes de errores. A continuación se repasan las posibles soluciones a esta necesidad, la escogida para llevar adelante este proyecto, la forma de integración en la aplicación y el método de funcionamiento para el trabajo con el servicio.

3.7.1. Alternativas disponibles

Google proporciona su propio sistema de reporte de errores integrado en Google Play. Si una aplicación ha sido instalada a través de su plataforma, en el momento del error se muestra un mensaje de alerta que ofrece dos opciones: aceptar para cerrar sin realizar ninguna acción, o reportar el informe de error.

[8]En el caso de Faborez, es `http://stark-scrubland-5936.herokuapp.com/`

No obstante, pocos usuarios reportan el fallo, por la sensación de pérdida de privacidad que supone (se debe enviar una importante cantidad de información sobre el dispositivo) o por pura incomodidad de seguir los pasos.

Por los inconvenientes anteriores, se decidió analizar y pasar a utilizar un servicio alternativo no dependiente de Play Store. Existen numerosas opciones por Internet, fácilmente localizables por Google. Sin embargo, por la falta de conocimiento sobre el tema y las alternativas disponibles, una rápida comparación llevó a la elección: Bugsense. Las características eran similares entre unos y otros servicios, pero este dispone de un plan básico totalmente gratuito, apropiado para este proyecto.

3.7.2. Integración

Bugsense se integra con la aplicación mediante dos sencillos pasos. Por un lado, se debe instalar una librería de Java que la propia empresa facilita para ser descargada (o mediante un repositorio Maven). Después, se debe añadir una línea de código en el arranque del programa, que tiene como único parámetro un *token* que identifica al desarrollador dentro de Bugsense.

De esta forma cuando la aplicación se cierra por un error, se guarda el informe antes del cierre, y este es enviado de forma automática cuando la aplicación vuelva a abrirse y exista una conexión a Internet activa. No es necesario ningún tipo de intervención por parte del usuario.

3.7.3. Método de funcionamiento

Los informes de errores recibidos se almacenan y se muestran al desarrollador, con la siguiente información, entre otros:

- La traza del error, con los archivos y número de línea dentro del código donde se han producido los errores.
- Toda la información del dispositivo: el modelo de móvil, la versión del sistema operativo Android, la cantidad de memoria RAM y otros datos relativos al estado, como las conexiones a Internet disponibles y la disponibilidad de la localización.
- Datos sobre la configuración del dispositivo, como el idioma establecido en el sistema operativo, o la cuenta de usuario activa en la aplicación, que el desarrollador debe encargarse de establecer en el propio programa.

Como la cantidad de errores que pueden recibirse para un mismo fallo puede ser grande, dependiendo del número de usuarios, estos se agrupan para la comodidad, y se muestran estadísticas relativas a los datos anteriores.

El método de funcionamiento con estos informes de errores es muy similar al trabajo con un sistema de *tickets* de soporte. El desarrollador debe corregir el fallo y después indicarlo en Bugsense, junto con la versión que lo arregla, para llevar un histórico. De esta manera, puede ignorar los informes de versiones en los cuales es conocido el error, o mostrar una alerta si se vuelve a repetir la incidencia en una versión posterior.

Capítulo 4.

Participación de los usuarios en el proyecto

Desde la concepción de la idea de Faborez, hasta la materialización de este en un proyecto definido y materializado existe un gran trabajo de recopilación de ideas y tomas de decisiones.

En muchos proyectos fin de carrera que se llevan a cabo en las Facultades de Informática, y probablemente en otras titulaciones, es el propio alumno el que especifica, diseña e implementa el producto a desarrollar y es quien evalúa los resultados, sin otra interacción con posibles clientes o usuarios que la que, eventualmente, ejerce el director del proyecto como *representante* de dicho colectivo.

Este proceso unipersonal, al que algunos profesores de Gestión de proyectos denominan «Proyectos Juan Palomo»[1], deriva en un producto con errores de diseño, carencias de funcionalidad y que para el resto de usuarios no es útil o agradable.

Por ello, la integración de los usuarios en el proceso de desarrollo del proyecto resulta esencial, si lo que se quiere lograr es un producto que realmente va a ser útil y popular. Son sus opiniones las que deben ser tenidas en cuenta en todo momento [23].

Este capítulo de la memoria describe la metodología seguida durante el trabajo con los usuarios, cuales han sido los criterios para su selección y categorización, las vías utilizadas para la obtención de *feedback* y su posterior procesamiento en el proyecto.

4.1. La muestra de *txapeldun*es

Los usuarios para la fase de pruebas deben cumplir con algunos criterios para que la experiencia, tanto la de usuario como la del desarrollador, sea satisfactoria y de resultados útiles.

[1] En referencia al dicho popular «Juan Palomo, yo me lo guiso, yo me lo como».

Además, este grupo de usuarios se dividirá en dos. El grupo mayoritario no pasará de ser un grupo estándar de usuarios, que utilicen y den vida a la aplicación y que mientras tanto reporten los fallos que vayan ocurriendo. Para escoger las personas de este grupo, se han analizado los siguientes parámetros:

- Localización geográfica habitual, para que converja una masa crítica de peticiones y usuarios dispuestos a responder. No es necesario que se forme un único núcleo, sino que puede haber varios si existieran las suficientes personas.

- Una gama amplia de dispositivos en los que se utiliza la aplicación aumenta la probabilidad de encontrar fallos y optimizar la aplicación para múltiples dispositivos. Entre otros, los factores a tener en cuenta serán: si es un teléfono o una tableta, versión del sistema operativo, tamaño de la pantalla, si utiliza un teclado táctil o es uno físico, etc.

- La personalidad de los propios usuarios es clave para obtener el tipo de *feedback* que se se quiera. Se descartan los perfiles de gente demasiado crítica (que pueden sobrecargar con comentarios no constructivos) o aquellos que pueden tender a llamar la atención mediante bromas en la aplicación. Son deseables aquellas personas que tengan un nivel técnico medio para proponer mejoras maduras y que estén habituados al uso de aplicaciones sociales en sus móviles. Por último, es conveniente contar con *tester*s de ambos géneros y de distintos perfiles lingüísticos.

- No obstante al punto anterior, también es interesante contar con perfiles «tópicos» que pueden dar lugar a circunstancias que planteen un reto para el proyecto no pensado hasta el momento. Por ejemplo, uno de estos usuarios podría realizar peticiones de favores no apropiados, generando así la necesidad de planificar cómo se deben gestionar estos hechos, antes de que ocurran en un público general.

De entre todos los usuarios de prueba, existirá un subconjunto con mayores funciones que el resto. Además del mero uso de la aplicación, tendrán un papel activo en el planteamiento de propuestas y mejoras, manteniendo una relación constante con el desarrollador. Estas personas se denominarán *txapeldun*es, en referencia a los campeones que antiguamente se disputaban en duelo en defensa y representación de otras personas.

Estos *txapeldun*es, además de cumplir con los puntos anteriormente enumerados, serán personas cercanas y de confianza, con la disponibilidad suficiente como para mantener reuniones regulares y su actitud deberá ser proactiva desde el inicio. La función de estas personas será la de visualizar la aplicación hacia el futuro y proponer mejoras hacia ese objetivo, fijarse en los detalles que el usuario corriente no percibiría y tomar las decisiones sobre el diseño en representación de otros usuarios de la aplicación.

4.2. Documentación a preparar

Los *tester*s necesitan están informados de su condición, de sus funciones y de lo que ocurre alrededor de la aplicación, por lo que se preparará la siguiente documentación para estos, que ayuden a cumplir los objetivos deseados [22]:

Guía del *txapeldun* Documento corto y preciso que describe, por una parte, la filosofía de Faborez y las características que tiene, junto con las exclusiones. Por otra, unas pautas de actuación como *tester*s: invitación a utilizar exhaustivamente la aplicación, a fijarse en las características que puedan ser mejoradas y a cómo transmitir el *feedback*.

Cuestionarios Las dinámicas para obtener el *feedback* no pueden ser completamente desatendidas, y las dinámicas deben estar dirigidas. Esta dirección se ha llevado a cabo utilizando unos cuestionarios preparados antes de las entrevistas, para ser entregados en ellas.

Los cuestionarios introducen inicialmente los cambios que ha habido desde la última entrevista para poner en contexto al usuario. Posteriormente, se enumeran las acciones que deberían realizar junto con las preguntas sobre estas características, o notas para centrar la atención en puntos del diseño.

Actas de *feedback* Los comentarios, aportaciones y cualquier otro dato interesante aportado se anota en actas fechadas. Al ser un documento directo del *feedback*, es importante no realizar filtro de los comentarios, más allá de un mínimo, y recibir todos los aportes tal cual sean transmitidos por el *betatester*.

Propuestas de mejora Del *feedback* en bruto recibido de los usuarios, tras analizar su viabilidad, unificar las similares y adaptarlas a las posibilidades, son exportadas en un documento que recoge las propuestas de mejora que se llevarán a cabo en un futuro. Este documento es dinámico, ya que se irán añadiendo a la lista nuevas propuestas, se descartarán algunas anteriores o nuevas ideas pueden superar y reemplazar a las anteriores.

4.3. Metodología de interacción

La participación de los usuarios en el proceso de desarrollo requiere de un procedimiento dedicado y de planificación propia, teniendo claro en cada interacción la información que se quiere obtener de ellos.

Esta es la metodología seguida para integrar los comentarios de los *txapeldun*es durante el desarrollo de este proyecto. El proceso pasa por las fases de introducción, *feedback* y de notificación posterior sobre el progreso.

4.3.1. Introducción a la aplicación

La interacción comienza en el momento que el usuario se topa con la aplicación. Los *txapeldun*es son informados en el inicio sobre qué es Faborez y sobre cuales serán sus funciones como *tester*s.

Como muchos de los comentarios de los usuarios surgen en el primer contacto y pueden pasar desapercibidos, se debe estar presente desde el inicio. Les es repartido, paralelo a la instalación, el documento «Guía de los *txapeldun*es».

4.3.2. Feedback sobre la usabilidad

Dadas las funcionalidades de la aplicación, se les invita a que pongan en marcha cada uno de los usos de caso, pero sin las instrucciones para ello. Se toma nota de la rapidez con la que los usuarios actúan para llevar a cabo estas tareas, y fijándose en los sitios donde puedan tener dudas sobre cómo avanzar.

Los cuestionarios serán los que encaminen este procedimiento, asegurando que incluyan apartados relativos a aquellos diseños de la aplicación que han sido decididos por el desarrollador y no por los usuarios.

4.3.3. Notificación de novedades

Todos los usuarios notarán que la aplicación es automáticamente actualizada en el momento que se suba una nueva versión, pero no obstante no conocerán las novedades que esta versión pueda traer consigo.

Para cada actualización significativa los *txapeldun*es serán informados acerca de la novedad por dos motivos. Primero, para mantener su nivel de implicación haciéndoles parte de toda la información. Y segundo, para que prueben las nuevas funcionalidades y sea posible depurar los posibles errores.

4.4. Canales de comunicación

Los *txapeldun*es deben tener vías de comunicación directas con el desarrollador y desde el propio teléfono móvil. Es más, es interesante que puedan hablar entre ellos y proporcionarse información sin contar con el desarrollador. Para esta tarea se ha utilizado la aplicación Telegram, creando un grupo de contactos en esta plataforma. Además de escribir mensajes de texto, permite enviar capturas de pantalla y cualquier otro archivo multimedia de forma sencilla.

Faborez es también un medio excelente de comunicación con los usuarios. Ante la necesidad de que los *txapeldun*es realicen algunas tareas dentro de la aplicación, puede ser transmitido como una petición de *favor* más. En la misma línea, las peticiones pueden contener un aliciente en forma de recompensa a cambio de, por ejemplo, responder a la demanda.

Capítulo 5.

Arquitectura del servicio

Para entender el funcionamiento del servicio Faborez es necesario explicar la arquitectura de la solución final, identificando cada una de sus partes y la forma en la que estos interaccionan para funcionar. En la figura 5.1 se muestra de forma gráfica el conjunto del sistema.

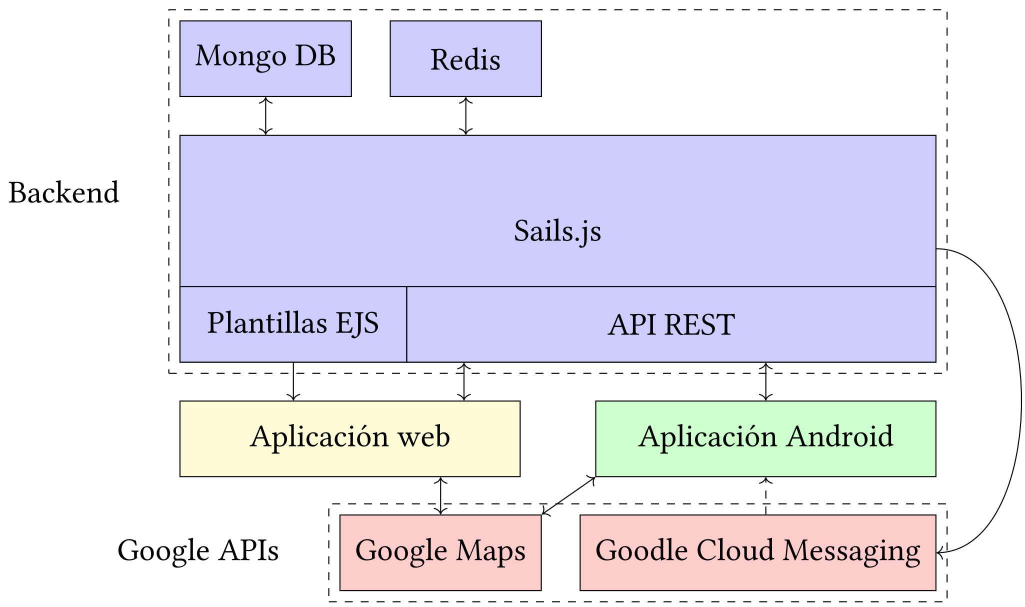

Figura 5.1.: Arquitectura de Faborez

El servidor de la aplicación juega de punto central de la aplicación, que realiza la labor de almacenar los datos en la base de datos y ejecutar la lógica de la aplicación. A este se conectan los distintos clientes instalables mediante la api REST, o directamente mediante un navegador a la *webapp*.

Este servidor se encuentra alojado en Heroku, que ejecuta el *backend* y proporciona los servidores de MongoDB y Redis. Como se ha mencionado el apartado 3.6, Heroku utiliza Amazon AWS como base.

El *backend* utiliza dos servicios de Google, Plus y GCM, para la autenticación de usuarios y el envío de notificaciones a móviles respectivamente.

En cuanto a los clientes, existen la aplicación web y la aplicación para Android. La primera está fuertemente ligada al *backend*, en el sentido de que parte de su código sale del procesamiento de las plantillas del servidor. La lógica en el cliente está implementada mediante Backbone.js y Marionette.js para algunas páginas, en las cuales los datos son obtenidos del servidor utilizando la misma API que utiliza el cliente Android.

El cliente Android hace un uso completo de la API REST del servidor, del cual descarga todos los datos necesarios durante el funcionamiento, que no se almacenan permanentemente. El cliente Android recibe notificaciones a través de GCM. Otra API de Google utilizada es la de Maps, que le permite mostrar los mapas.

En resumen, la arquitectura es del tipo **cliente–servidor**, en el cual el último tiene un peso mayor que los otros al tener este el estado general de los datos en cualquier momento. La comunicación es del tipo *pull* en la mayoría de los casos: los clientes realizan las peticiones por iniciativa de estos. La excepción es el envío de notificaciones desde GCM a los clientes Android, que se realiza mediante la estrategia *push*.

5.1. Comunicación entre componentes

La comunicación entre los distintos componentes se realiza mediante los protocolos HTTP (mayoritariamente) y el proporcionado por GCM. En el primer caso, la codificación de los datos se puede dar en los formatos HTML o JSON.

Los datos codificados en formato JSON tienen la estructura que se muestra en el extracto 5.1. Se envían todos los datos relativos a la clase, y se embeben también los objetos ligados a estos, en este caso el usuario que envió la petición y la categoría. Eso último permite simplificar la implementación de los clientes y reducir significativamente el número de peticiones al servidor.

La transferencia de datos mediante GCM es distinta, ya contiene algunas restricciones en cuanto al tipo y la cantidad de datos que se pueden enviar [9]. Por un lado, el total de los datos no debe superar los 4 KiB, y por el otro no es posible enviar tipos de datos complejos, deben ser solamente tipos de datos simples.

Por lo tanto, el *backend* de Faborez opta por la estrategia de indicar al dispositivo el tipo del contenido que se ha creado junto con su identificador, en la forma que muestra el extracto 5.2. Así, el cliente conoce con exactitud la URL de los datos que debe descargar y actuar posteriormente.

```
{
  ”title”:”Un paraguas.”,
  ”description”:”Paraguas para no mojarme.”,
  ”latitude”:43.3164306,
  ”longitude”:-1.984446,
  ”categoryId”:”52ffb6e789969d0b004d4886”,
  ”expiresAt”:”2014-03-03T17:06:23.389Z”,
  ”userId”:”52b2d7664ebade0b00c2063f”,
  ”createdAt”:”2014-03-02T17:06:23.389Z”,
  ”updatedAt”:”2014-03-02T17:06:23.390Z”,
  ”id”:”5313650f36dc1d0b009f2bbf”,
  ”user”:{
    ”name”:”Goiatz Irazabal”,
    ”email”:”goiatz@irazabal.com”,
    ”createdAt”:”2013-12-19T11:24:22.732Z”,
    ”id”:”52b2d7664ebade0b00c2063f”,
    ”avatarUrl”:”http://www.gravatar.com/avatar/dae2f01bdbd0612247c398ecbf854aba?s=128”
  },
  ”category”:{
    ”title”:”Prestar objeto”,
    ”id”:”52ffb6e789969d0b004d4886”
  },
  ”status”:”expired”
}
```

Extracto de código 5.1: Petición de favor codificada en formato JSON

```
{
  ”registration_id”:”APA91bHun4MxP5egoKMwt2KZFBaFUH-1RYqx...”,
  ”data”:{
    ”type”:”request”,
    ”id”:”5313650f36dc1d0b009f2bbf”
  }
}
```

Extracto de código 5.2: Mensaje de GCM con *payload* incrustado

5.2. Alertas de peticiones

Cuando un usuario realiza una petición, se realiza un procedimiento que implica a varias partes de la arquitectura. El *backend* filtra a los dispositivos utilizando las coordenadas de la petición, envía a GCM sus identificadores y el identificador de la petición, y este servicio es el encargado de realizar las notificaciones mediante el método *push*. En la figura 5.2 se ve de forma gráfica el flujo de los datos.

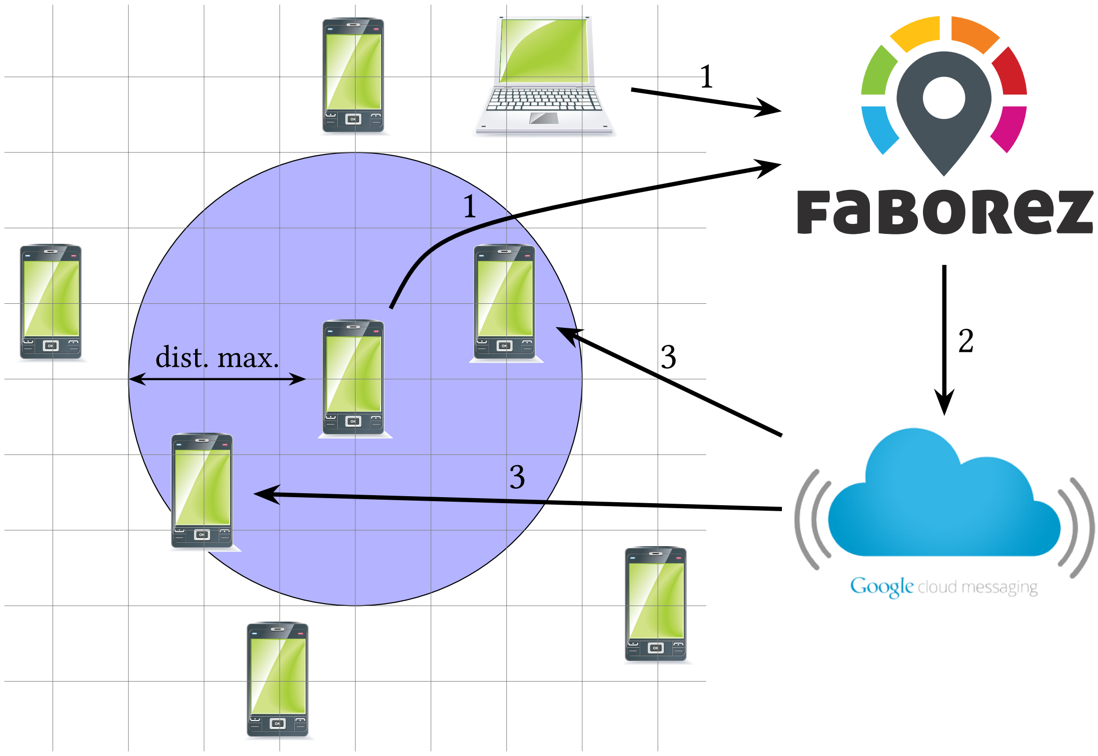

Figura 5.2.: Envío de alertas a dispositivos cercanos

5.3. Autenticación de usuarios

Faborez no implementa un sistema propio para la autenticación de sus usuarios. Es decir, no se guardan credenciales de acceso que luego el usuario utiliza para identificarse, sino que se delega esta tarea en el servicio externo Google+. Existen alternativas como Twitter, Facebook o incluso Github, pero se ha optado por primar la homogeneidad de los servicios externos utilizados.

Este servicio se basa en el protocolo OAuth 2.0, en el cual Faborez toma el rol de cliente [16, sec. 1.1]. Primero, Faborez redirige al usuario a una pantalla dentro de Google en la que se le muestran los permisos solicitados[1] y podrá conceder el acceso. Este acceso se materializa en un ***token* de autenticación** que es transferido a Faborez. A continuación, el *backend* canjea este *token* de autenticación por un ***token* de acceso**, con el que ya puede descargar la información solicitada previamente.

El proceso en los clientes web y Android es similar en cuanto a los pasos. La diferencia está en que, en el cliente web, el flujo del proceso se controla con redirecciones HTTP, mientras que el cliente Android hace uso de la librería *Google Play Services*. En la figura 5.3 se muestra este proceso de forma gráfica.

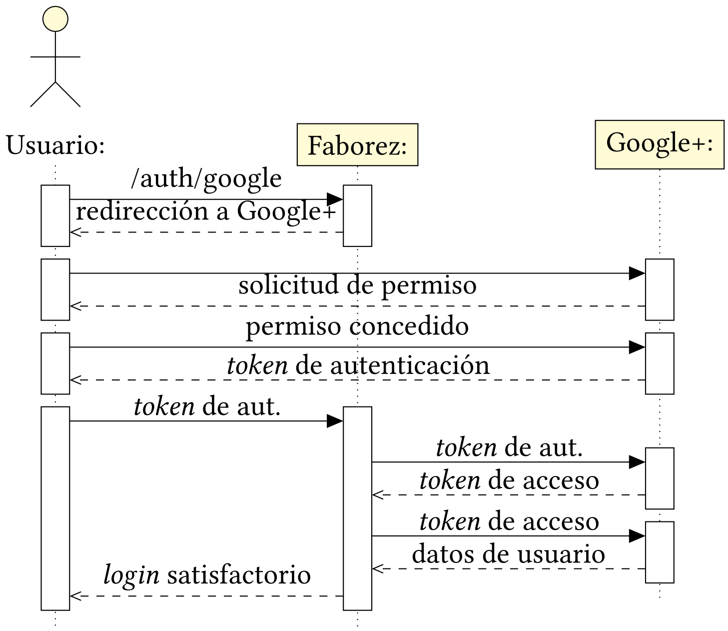

Figura 5.3.: Flujo de autenticación mediante Google+

El cliente web recibiría como resultado una *cookie* con la información de la sesión, pero en el cliente Android, en cambio recibe unos *token*s de acceso y de refresco creados por el *backend* de Faborez. De esta manera, se forma otro conjunto Oauth, en el que el *backend* pasa a ser el *resource owner* y la aplicación el cliente. Para el posterior refresco, se produce el flujo de la figura 5.4.

[1]Faborez solicita los permisos para acceder a la información básica (nombre y avatar) y la dirección de correo electrónico.

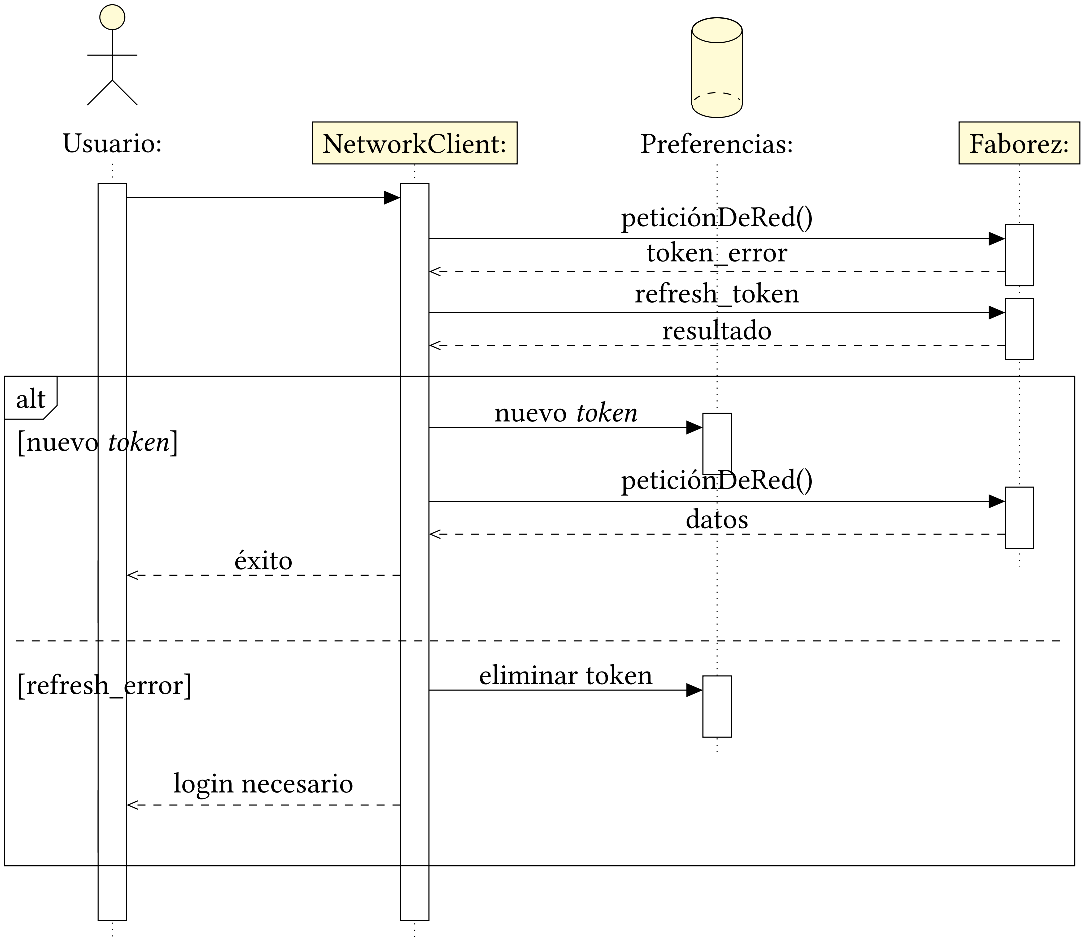

Figura 5.4.: Refresco de *token*s en el cliente Android

Capítulo 6.

Servidor *backend*

El servidor de Faborez es la pieza central del servicio: implementa la lógica de negocio al completo, almacena los datos de forma permanente y los clientes los obtienen de este.

Como se ha explicado en el apartado 3.2, este servidor se encuentra implementado en Sails.js, por lo que sigue la estructura definida por este *framework*, a saber:

- La definición de los modelos, que describen sus atributos, la forma en la que se guardan en la base de datos y algunos métodos de ayuda para trabajar con sus datos.
- Los controladores, en los cuales se implementa toda la lógica de negocio. Tratan las peticiones enviadas y devuelven los resultados en el formato adecuado (HTML o JSON).
- Las plantillas, en formato EJS, para enviar los resultados en formato HTML.
- Los distintos archivos de configuración, entre los cuales los más importantes son:
  - Las políticas de acceso que definen la autenticación y la autorización requerida para ejecutar una acción de un controlador determinados.
  - Las rutas de acceso, que unen las distintas URL de la API con su respectiva acción en un controlador.
  - La información relativa a las bases de datos MongoDB, que son su dirección y credenciales de acceso y Redis, con una configuración similar.
  - El *hook* a través del cual Sails.js permite integrar *middleware*, que en su caso se trata del sistema de autenticación.
- En otra carpeta también se encuentran las librerías Javascript para la aplicación web que el navegador directamente descarga.
- Utilidades para algunos de los casos de uso de la lógica de negocio, incluidos el envío de notificaciones con GCM y tareas periódicas para la limpieza de la base de datos.

Este capítulo describe el modelo de los datos almacenados en el *backend*, la lógica de negocio junto con su forma de estructuración y algunos detalles relativos a la configuración del servidor.

6.1. Modelo de datos

En la figura 6.1 se muestra diagrama de clases que el *backend* de Faborez maneja. Cada uno de estos modelos se corresponde con una colección en la base de datos MongoDB (ver apartado 3.1). Las nueve clases que conforman el modelo son: `User`, `Request`, `Category`, `Report`, `Message`, `Device`, `Token` y `RefreshToken`.

`User` Representa a los usuarios de Faborez, que se identifican mediante su dirección de correo electrónico.

`Request` Es la petición de favor, que contiene título, descripción, coordenadas de geolocalización, las fechas de creación y caducidad, además de las referencias a la categoría y el usuario. Existe una referencia que puede ser nula, que es aquella que indica qué respuesta fue la que satisfizo a la petición, en su caso.

El método `getStatus()` depende de este último aributo y de la fecha de caducidad, pudiendo devolver tres posibles valores (ver implementación en el extracto 6.1): resuelto, caducado o abierto.

`Category` Representa a la categoría de las peticiones. Aunque los valores posibles se guarden en la base de datos, estos no cambian frecuentemente.

`Report` Denuncia por abuso sobre una petición en concreto.

`Reply` Hilo de conversación de respuesta perteneciente a una petición y a un grupo de ayudantes concreto. El atributo `deleted` marca como eliminada este hilo de respuesta, sin que realmente se borre de la base de datos.

`Message` Cada uno de los mensajes del hilo de conversación. Contiene una referencia al usuario autor del mensaje y al `Reply` correspondiente.

`Device` Identifica a los dispositivos móviles, por ahora solamente Android, registrados por cada usuario. Se guarda un identificador (`deviceId`) proporcionado por GCM y la geolocalización enviada. Estos datos son utilizados para enviar las notificaciones de petición a las personas cercanas.

`Token` y `RefreshToken` Son los *token*s utilizados para la autenticación de los dispositivos móviles, tal y como indica el estándar OAuth 2.0 [16]. Para ambos, se guarda la caducidad, el *token* en sí y la referencia al usuario al que pertenecen.

6.2. Lógica de negocio

La lógica de negocio del servidor de *backend* se implementa en los *controladores* de Sails.js. Los controladores están formados por acciones, que se agrupan según la clase de datos que tratan o su funcionalidad. Estas acciones, a su vez, van ligadas a una URL concreta.

```
function getRequestStatus() {
  if (this.finalReplyId) {
    return 'resolved';
  } else if (moment(this.expiresAt).isBefore()) {
    return 'expired';
  } else {
    return 'open';
  }
}
```

Extracto de código 6.1: Método `getStatus()` de la clase `Request`

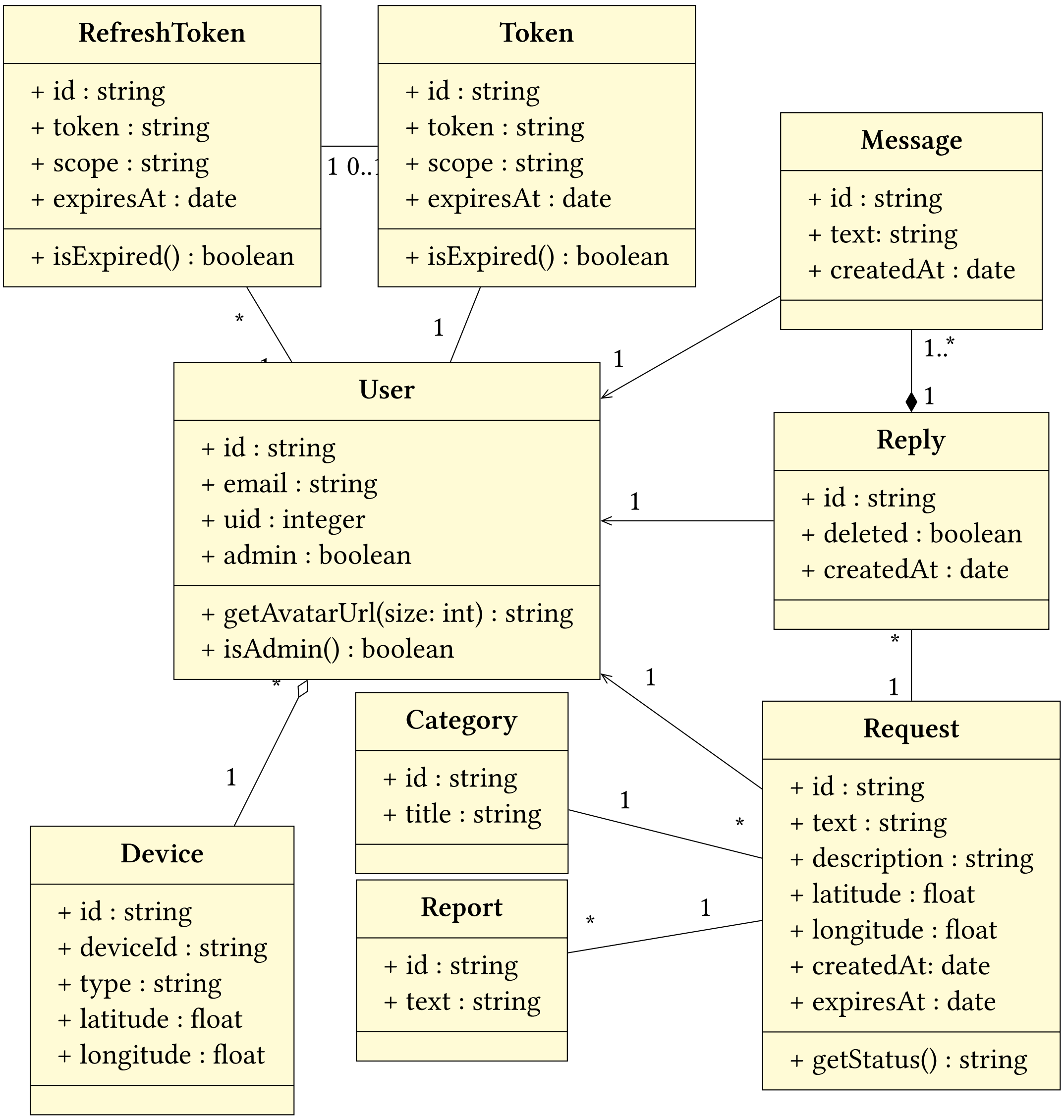

Figura 6.1.: Modelo de datos del servidor

Estos controladores, según su naturaleza, pueden ser agrupados en tres categorías distintas:

1. Los controladores de la API REST, en los que cada uno está vinculado a una clase de datos concreta, y provee acciones de creación, lectura, actualización y eliminación de los distintos objetos. La mayoría de los controladores se agrupan en esta categoría, y su URL tiene el prefijo `/api/v1/`.

2. Controladores relativos a la seguridad, encargados de la creación de los usuarios y de su autenticación, ya sea con el sistema de Google o con los *token*s que el propio sistema proporciona.

3. El controlador `FrontendController`, que contiene las rutas por las que se accede mediante la aplicación web. A diferencia del resto, este controlador siempre tiene como salida texto HTML.

En el cuadro 6.1 se muestran todas las acciones agrupadas por controlador, junto con sus rutas y el nivel de acceso requerido. Se encuentran en ella las acciones más relevantes, y se descartan las menos importantes, aún cuando estén implementadas.

6.3. Configuración del servidor

El trabajo de implementación se realiza íntegramente en el ordenador local, y es ahí donde es ejecutado en un principio para ser probado. Después es subido a Heroku, poniéndolo en producción. En el ordenador local, se utiliza la base de datos instalada en el propio ordenador, que aparte de contener datos distintos, los parámetros de conexión son distintos a los del servidor de producción.

Resulta que surge un problema de configuración. Lamentablemente, Heroku no permite editar alguno de los archivos ya que se vale de Git y todo el código es desplegado en bloque.

Por otro lado, una cuestión a tener en cuenta es que los parámetros de conexión del servidor de producción deben mantenerse privados, por lo que no deben estar presentes en el código al estar este visible en el servidor de control de versiones. De la misma forma, también es conveniente que los parámetros locales tampoco se encuentren en el repositorio, ya que puede haber distintos desarrolladores trabajando al mismo tiempo, con distintas configuraciones cada uno.

Afortunadamente, es posible sortear ambos problemas con dos métodos: por un lado las variables de entorno, presentes en todos los sistemas operativos y sus programas, y por otro el fichero de configuración `local.js` que Sails.js habilita en su sistema de ficheros.

Ruta	Controlador	Acción	Acceso

`GET /`
`GET /requests`
`GET /request/:id`
`GET /messages/:id`
`GET /profile/:id`
`GET /profile/me`
	FrontendController	index
		requests
		request
		messages
		profile
		myProfile
			Usuario

`GET /auth`
`GET /google`
`GET /google/callback`
	AuthController	index
		google
		google/callback
			Público

`GET /logout`		logout	Usuario
`GET /api/v1/user/:id?` `GET /api/v1/user/me`	AuthController	find me	Usuario

`GET /api/v1/request/:id?`
`POST /api/v1/request`
`PUT /api/v1/request/:id`
`DELETE /api/v1/request/:id`
	RequestController	find
		create
		update
		destroy
			Usuario

`GET /api/v1/category/:id`	CategoryController	find	Usuario

`GET /api/v1/reply/:id?`
`POST /api/v1/reply`
`DELETE /api/v1/reply/:id`
`POST /api/v1/reply/:id/accept`
	ReplyController	find
		create
		destroy
		accept
			Usuario

`GET /api/v1/message/:id?` `POST /api/v1/message`	MessageController	find create	Usuario
`GET /api/v1/report/:id?`	ReportController	find	Admin
`POST /api/v1/report`		create	Usuario
`POST /device/report`	DeviceController	report	Usuario

`GET /oauth/authorize`
`POST /oauth/token`
`POST /oauth/google`
	OauthController	authorize
		token
		google
			Público

Cuadro 6.1.: Las acciones y sus respectivas rutas

`local.js` Este fichero es el último de entre los archivos de configuración que Sails.js lee. Esto permite que sobreescriba cualquier valor definido en otro fichero. Posteriormente, se utiliza el archivo `.gitignore` para indicar que el archivo `local.js` no debe ir al repositorio de código.

El extracto 6.2 muestra un ejemplo de este fichero de configuración, en el que se configura el servidor Redis para almacenar la sesión de los usuarios en el entorno local.

```
module.exports = {
  session: {
    adapter: 'redis',
    host: 'localhost',
    port: 6379,
    db: 0,
    pass: null
  },
  port: process.env.PORT || 1337,
  environment: process.env.NODE_ENV || 'development'
};
```

Extracto de código 6.2: Archivo de configuración `local.js`

Variables de entorno Heroku permite utilizar variables de entorno en las que introducir los parámetros de configuración. Estos no se escriben en el código, si no que se acceden desde él. El extracto 6.3 es un ejemplo de la configuración de la base de datos, en el que utiliza la variable de entorno `MONGOHQ_URL`, si existe, y un valor por defecto si no está presente.

```
module.exports.adapters = {
  'default': 'mongo',

  mongo: {
    module: 'sails-mongo',
    url: process.env.MONGOHQ_URL || 'mongodb://localhost:27017/faborez'
  },
};
```

Extracto de código 6.3: Configuración de la base de datos mediante variables de entorno

Capítulo 7.

Cliente Android

La aplicación móvil para Android es el cliente preferencial[1] para acceder a Faborez. En ella se implementan todos los casos de uso que el servicio dispone.

Como se ha mencionado en el apartado 3.3, la aplicación se ha implementado mediante Java, utilizando la API nativa de Android. Por lo tanto, los casos de uso se dividen en las distintas partes de la interfaz (llamados *Activity*) y los servicios en segundo plano (llamados *Service*s), que se describen más adelante.

7.1. Modelo de datos

Siguiendo la arquitectura basada en el servidor (capítulo 5), el cliente Android no almacena de forma permanente los datos de la lógica de negocio que genera o descarga del servidor, y solo existen mientras se encuentran en memoria. Esta es una decisión de diseño que tiene como objetivo evitar la inconsistencia de datos entre cliente y servidor.

Los datos guardados son los relativos a la autenticación de los usuarios y los ajustes. El método para el almacenamiento de estos es la clase `SharedPreferences` de Android [14], que almacena los datos en formato XML. En el cuadro 7.1 se muestran los datos que son almacenados.

No obstante, la aplicación Android sí tiene implementadas las clases necesarias para trabajar con los datos que se descargan del servidor, en forma de *Plain Old Java Object*s (POJOs). La librería Jackson se utiliza para convertir las cadenas JSON en estos objetos POJO. Las clases se muestran en la figura 7.1.

[1]Debido a algunas carencias del cliente web, como las notificaciones, que hacen que tenga una funcionalidad más reducida.

Archivo	**Campo**	**Descripción**
`userconfig.xml`	user	Perfil del usuario en formato JSON
	categories	Categorías sincronizadas del servidor en formato JSON
`faborez_preferences.xml`	default_expirity	Caducidad por defecto de las peticiones
	notifications_new_message	Si se deben enviar notificaciones
	notifications_new_message_ringtone	Sonido de las notificaciones
	notifications_new_message_vibrate	Si la vibración se encuentra activada
	custom_language	Idioma por defecto
`NetworkPrefs.xml`	access_token	*Token* de acceso
	refresh_token	*Token* de refresco
	token_expiration	Fecha de caducidad del *token* de acceso

Cuadro 7.1.: Preferencias almacenadas por el cliente Android

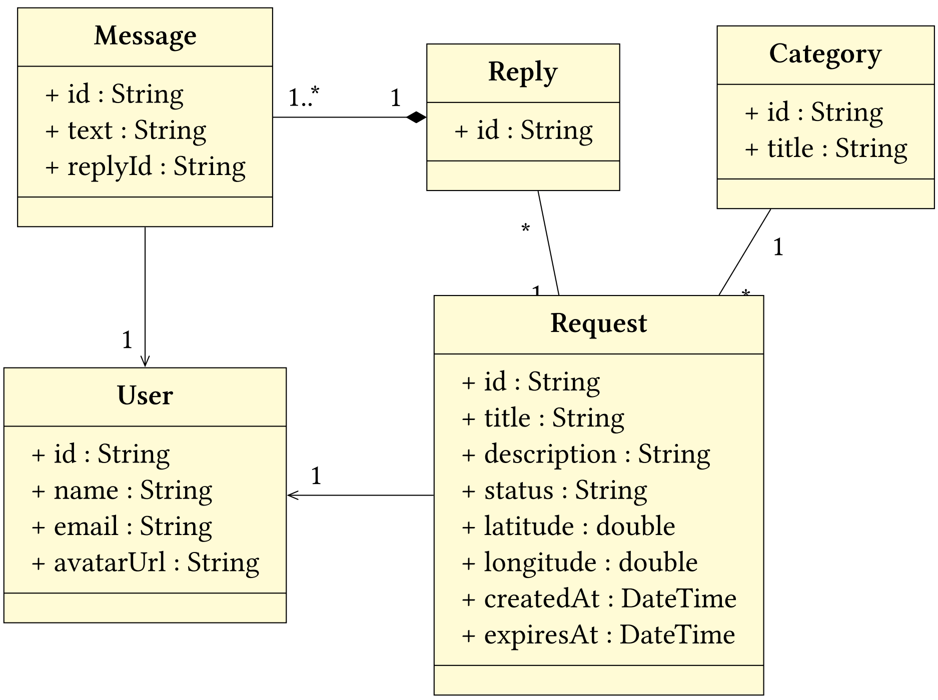

Figura 7.1.: Modelo de datos en memoria del cliente Android

7.2. Casos de uso

Los casos de uso de la aplicación de Android se dividen en dos categorías. Por un lado, están todas aquellas operaciones que el usuario ejecuta de forma explícita durante el uso mediante la interfaz gráfica. Por otro, son aquellas operaciones que se ejecutan de forma automática en segundo plano.

7.2.1. Interfaces de usuario

Las interfaces de usuario de Android se implementan mediante las llamadas *Activities* que contienen toda su lógica y controlan su ciclo de vida. Estas interfaces a su vez pueden estar compuestas por varias *Fragment*s, que son subinterfaces reutilizables con su propio ciclo de vida. A continuación se presentan las siete *Activities* de Faborez: `SplashActivity`, `MainActivity`, `MakeRequestActivity`, `ShowRequestActivity`, `ReplyActivity`, `UserActivity` y `SettingsActivity`. En la figura 7.2 se muestra la navegación entre ellas.

SplashActivity

Es la interfaz que se muestra al iniciar el programa por primera vez. Se muestra el logo de la aplicación, junto con un icono de Google+ para iniciar sesión. Este botón comienza la secuencia de autenticación, primero con Google y después con el *backend* de Faborez, para a continuación redirigir al usuario a `MainActivity`.

MainActivity

El punto de partida de todas las acciones que se pueden realizar en el cliente de Android de Faborez. En la parte inferior se encuentra el botón que permite realizar una petición, redirigiendo a `MakeRequestActivity`.

Por otra parte contiene dos menús laterales uno a cada lado. En el izquierdo, en coordinación con el mapa, se pueden encontrar las peticiones de favor cercanas al usuario. El de la izquierda es un menú propiamente dicho, que cambia la información que `MainActivity` muestra (cambiando el *Fragment*), o conduce a la interfaz de ajustes `SettingsActivity`.

Los *Fragment* que la parte central puede mostrar son las siguientes:

MapFragment Muestra el mapa de las peticiones cercanas, así como el botón inferior para realizar peticiones. En el menú se muestra como «Inicio».

UserRepliesFragment En el cual se encuentran las respuestas a las peticiones del usuario actual. Se denomina en el menú como «Mis respuestas».

UserRequestsFragment Este es el histórico de peticiones del usuario. Los colores indican el estado de la petición. En el menú se muestra como «Mis peticiones».

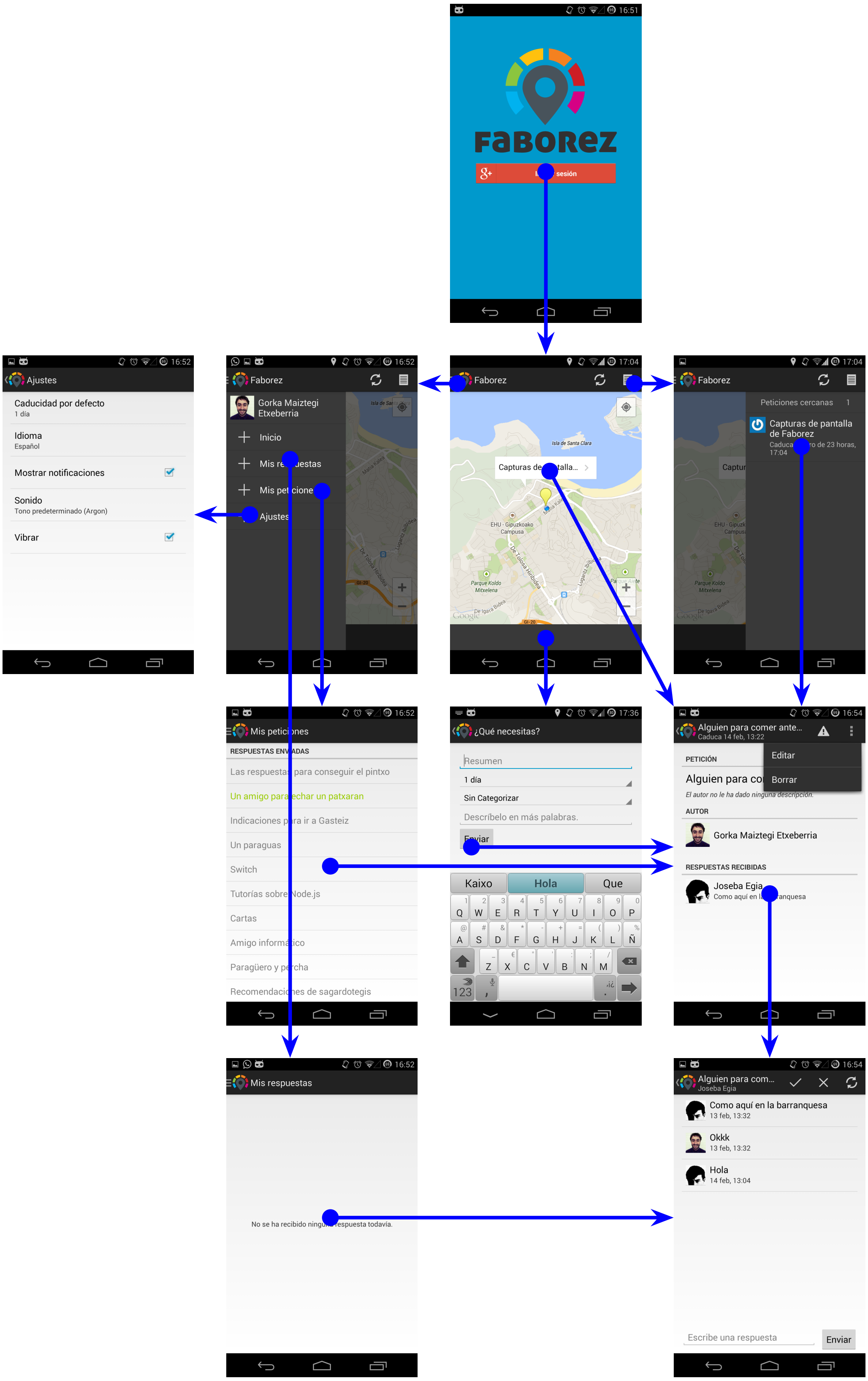

Figura 7.2.: Las pantallas de la interfaz de usuario y el movimiento entre ellas

MakeRequestActivity

Esta interfaz se vale de `RequestFormFragment` para mostrar el formulario de creación de una petición. Si la creación tiene éxito, se abre una instancia de `ShowRequestActivity` que muestra sus datos.

ShowRequestActivity

Esta interfaz muestra los detalles sobre una petición de favor. Por un lado su título, descripción y caducidad, por otro un enlace al perfil del usuario y por último las respuestas que la petición ha recibido. Si la petición que se muestra es de otro usuario, se muestra un enlace para escribir una respuesta a la petición.

En el caso del autor, este enlace lleva a la interfaz `UserActivity`, y en el caso de las respuestas a `ReplyActivity`.

ReplyActivity

Es la actividad que muestra la ventana de conversación entre la persona que realiza la petición y la que responde. En él se pueden agregar más mensajes, además de cancelar el hilo de mensajes y aceptarlo como respuesta definitiva, en el caso del usuario que registró la petición.

UserActivity

Muestra el perfil del usuario, ya sea el activo u otro al que se ha llegado mediante una petición. Se muestra el nombre, el avatar y el histórico de peticiones del usuario.

SettingsActivity

A esta interfaz se llega mediante el menú lateral de la interfaz `MainActivity`. Muestra los parámetros que el usuario puede ajustar, a saber:

- Caducidad por defecto de las peticiones realizadas en el futuro.
- Idioma de la interfaz del usuario.
- Los ajustes relativos a las notficaciones: activarlos o desactivarlos del todo, la vibración y el sonido.

7.2.2. Servicios en segundo plano

Existen dos funcionalidades que se encuentran implementadas mediante servicios en segundo plano, y que se ejecutan al margen de que la aplicación se encuentre en pantalla o no.

Servicio de localización

Su funcionalidad es la de informar al *backend* de Faborez sobre la localización de los dispositivos de forma periódica, con la finalidad de enviar notificaciones solamente a los dispositivos cercanos a la petición.

`LocationService` es un servicio que comienza su ejecución tras el inicio de sesión o automáticamente cuando arranca el sistema operativo Android, si ya se había iniciado sesión anteriormente. En su inicialización realiza las siguientes tareas:

1. Inicializa el gestor de localización para recibir notificaciones periódicas de las coordenadas.
2. Registra la aplicación con GCM, del cual obtiene un código de identificación.
3. Este código es enviado al *backend* de Faborez, el cual lo almacena.

El servicio de localización sobre el que se basa `LocationService` es el *Location APIs* de *Google Play* [12]. Con la finalidad de ahorrar batería, se establecen unos límites sobre la periodicidad en la que se reciben notificaciones del cambio de localización, a la vez que se mantiene un mínimo de precisión. Los criterios son los siguientes:

- Como mínimo el dispositivo se debe haber desplazado **250 m** desde la anterior actualización.
- Se notificará cuando se desplace esta distancia, y en cualquier caso mínimamente **cada hora**.
- No obstante, como máximo se enviará una notificación cada **10 min** como máximo.

Cuando una notificación de *Location APIs* ocurra, el servicio enviará las coordenadas recibidas al *backend* de Faborez.

El servicio está programado para ejecutarse infinitamente, aunque el sistema operativo puede determinar que debe terminarse por varias razones. En el momento de la parada se desconecta el servicio de localización *Locaction APIs*, para dejar de recibir actualizaciones de la posición.

Servicio de notificaciones

El servicio de notificaciones es el encargado de recibir los mensajes entrantes de GCM y, en su caso, mostrar las notificaciones al usuario. A diferencia del servicio de localización, este servicio es de carácter pasivo, activándose por eventos externos y se cierra al terminar las tareas pertinentes.

El servicio se implementa en dos clases distintas: `GcmIntentService`, que contiene la mayor parte de la lógica y muestra las notificaciones, y `GcmBroadcastReceiver`, la clase auxiliar que recive los avisos de GCM y tiene como tarea activar el servicio principal.

El funcionamiento se activa cuando el *backend* envía una notificación al dispositivo, y sigue los siguientes pasos:

1. `GcmBroadcastReceiver` recibe la notificación del mensaje, activa el servicio `GcmIntentService` y se lo envía.
2. `GcmIntentService` comprueba si se trata de un mensaje correcto y decodifica sus contenidos: tipo de notificación e identificador del dato.
3. Según el tipo, conecta con el *backend* y descarga mediante su API REST todos los datos sobre el nuevo contenido.
4. Utilizando el contenido recién descargado, los datos del usuario y las preferencias de notificación, se construye una notificación de Android, que contiene los textos e imágenes correspondientes.
5. A no ser que estuviera desactivado, se muestra la notificación al usuario, mediante el servicio `NotificationManager`.

Una vez realizada la tarea, el servicio se desactivaría si no hubiera más mensajes que tratar, aunque este es un proceso realizado internamente por el sistema operativo, por lo que su funcionamiento es transparente al usuario.

Capítulo 8.

Aplicación web

La interfaz en forma de página web podría considerarse la forma secundaria de acceder a Faborez, debido a que aunque pueden realizarse casi todas las operaciones del cliente Android carece de funcionalidades como las notificaciones. Además, la necesidad de configurar el navegador, o de dar una dirección de forma manual, dificultan la obtención de la localización del usuario. De esta forma, se pierde gran parte del valor que tiene el servicio.

8.1. Interfaz del usuario

A continuación se describen las distintas partes que tiene esta aplicación web, enumerando las partes de la interfaz web y detallando la funcionalidad que cada una ofrece. Se acompaña a cada una una captura de pantalla como ejemplo.

Pantalla de *login* La interfaz que se muestra inicialmente al acceder a la web de Faborez. Su única funcionalidad es comenzar el proceso de autenticación mediante Google+, tal y como se indica en el apartado 5.3 (ver figura 8.1).

Pantalla principal En ella se muestran, una vez obtenida la localización del usuario, las peticiones de favor más cercanas con los datos más relevantes y el enlace para acceder a las mismas. En otra pestaña, se encuentra el formulario para añadir nuevas peticiones (ver figura 8.2).

Visualización de petición Esta pantalla muestra todos los detalles sobre una petición en concreto. Si la petición es del propio usuario, se muestran todas las respuestas, junto con el botón de entrar a la conversación (ver figura 8.3).

Hilo de mensajes Al igual que en la aplicación Android, se muestran los mensajes intercambiados entre el usuario y la persona que respondió a la petición. Desde esta interfaz es posible responder con más mensajes o navegar a su perfil de usuario (ver figura 8.4).

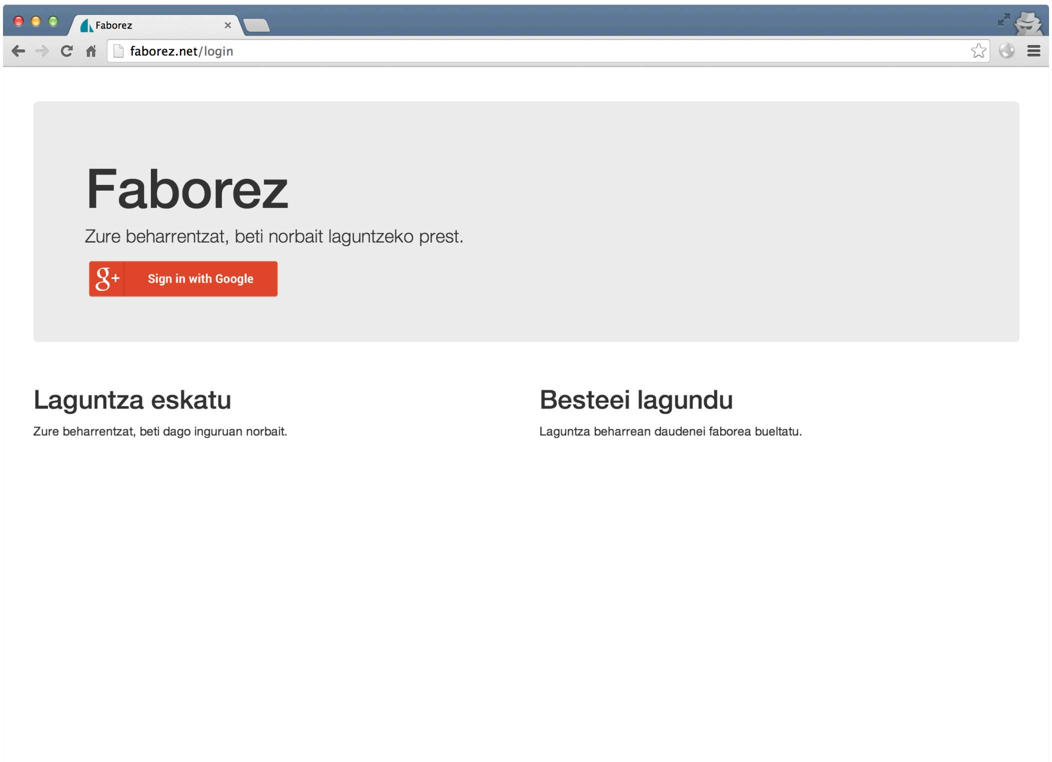

Figura 8.1.: Interfaz de *login*

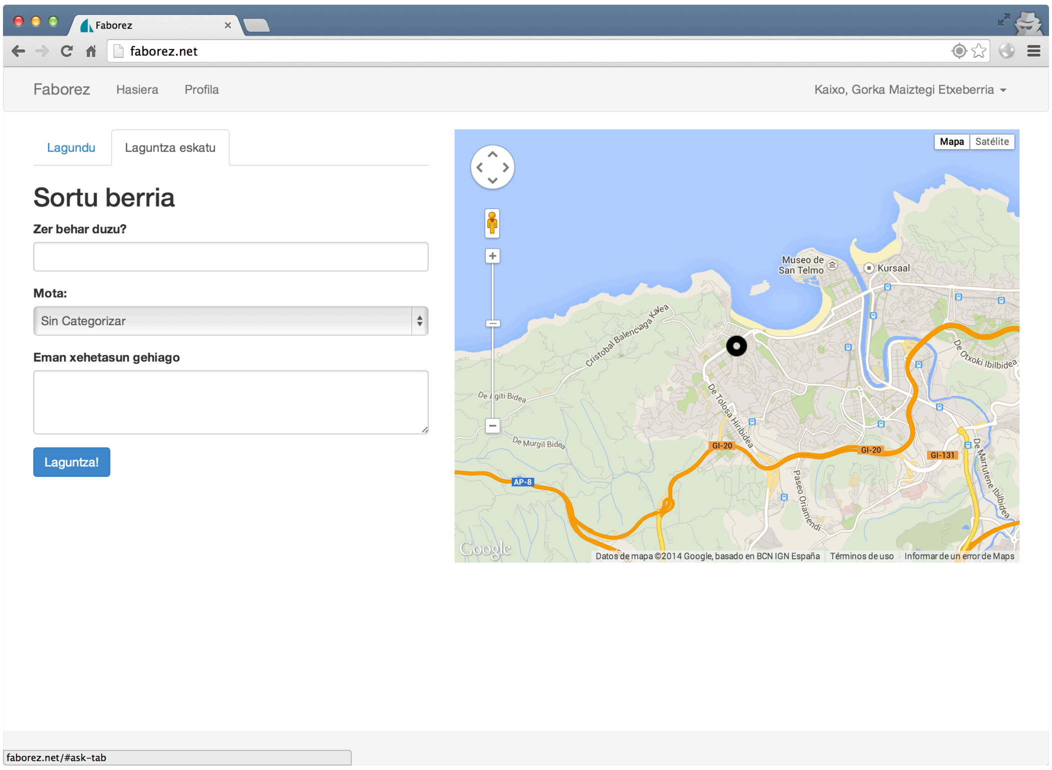

Figura 8.2.: Pantalla principal

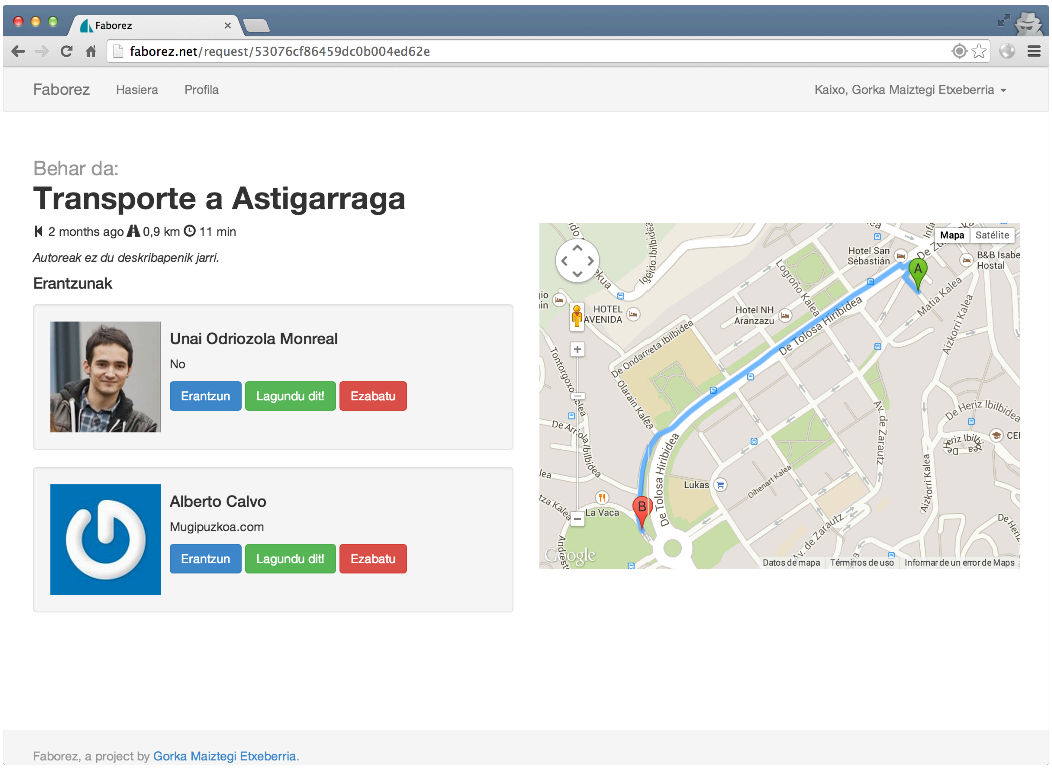

Figura 8.3.: Visualización de la petición

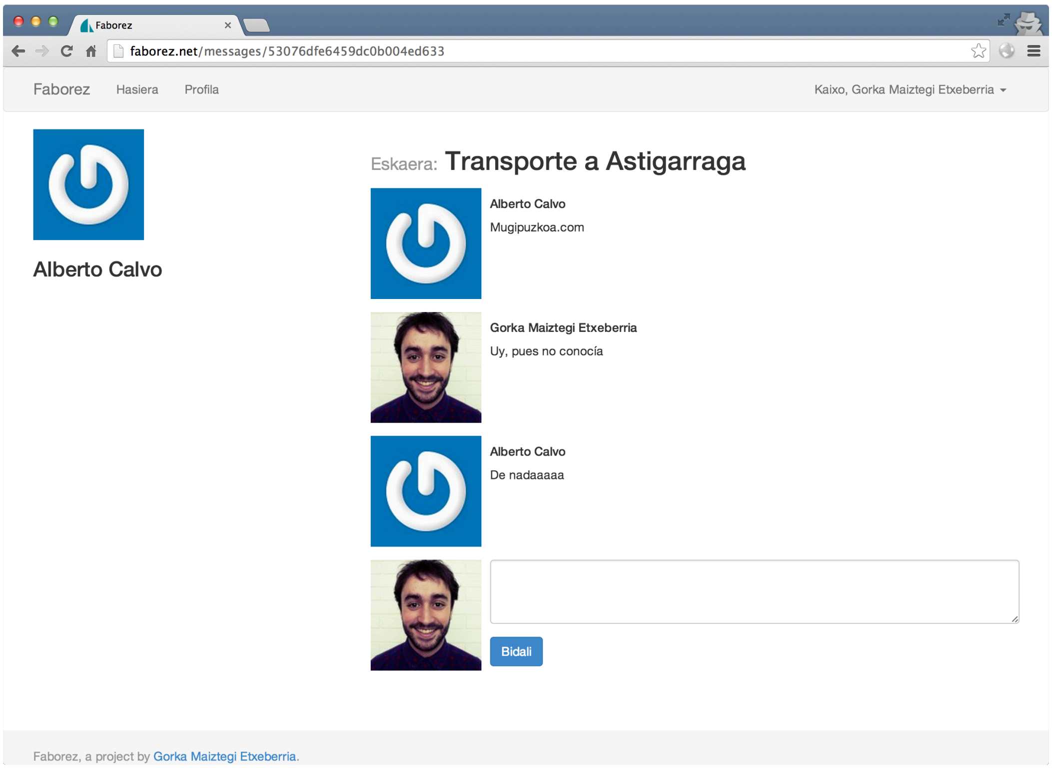

Figura 8.4.: Hilo de respuestas

Perfil de usuario En el perfil del usuario se puede visualizar información básica acerca de éste, junto con la actividad que ha tenido en Faborez, mostrando las peticiones de favor que ha realizado (ver figura 8.5).

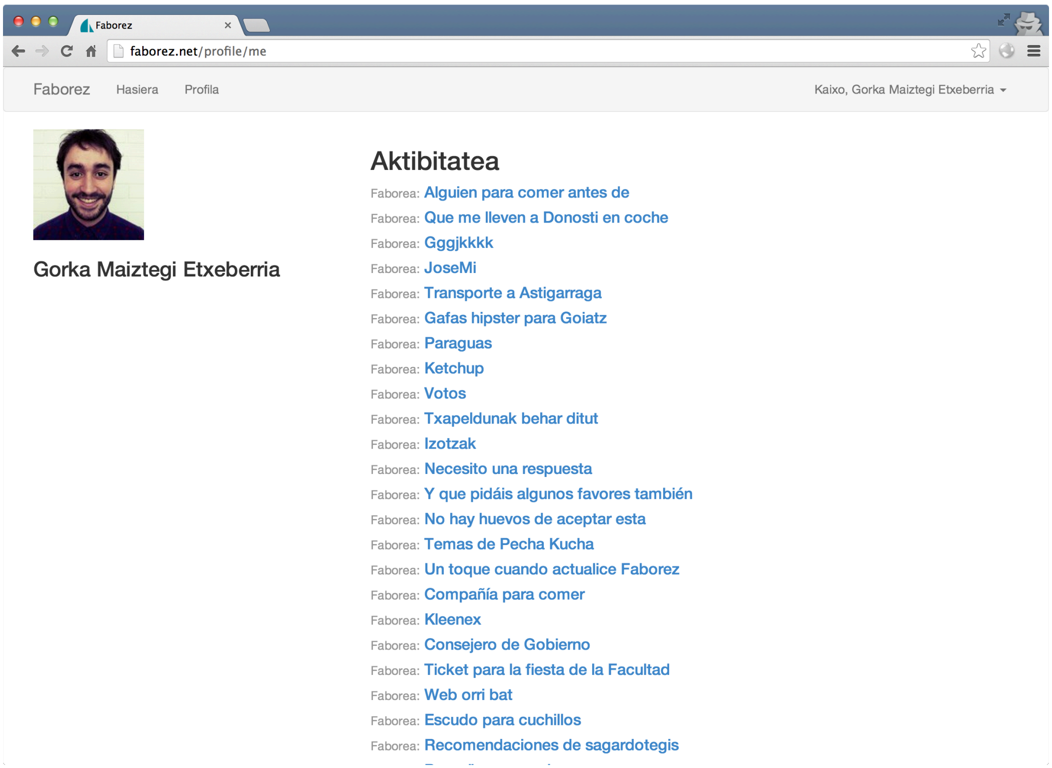

Figura 8.5.: Perfil del usuario

8.2. Utilización de APIs

La aplicación hace uso de dos librerías para completar el funcionamiento de algunas de las interfaces. Estas son la API de geolocalización [32] y el «API de rutas de Google» [8], utilizadas en dos partes distintas de la aplicación.

En el primer caso, la API de geolocalización es la forma estándar de HTML5 para obtener la localización. El usuario deberá dar permiso explícitamente al navegador, mostrándole una alerta para ello. Con la información obtenida, se envían las coordenadas al servidor para descargar las peticiones de favor más cercanas.

La utilidad de Google es utilizada en la interfaz de detalle de la petición, valiéndose de la localización del usuario, muestra en el mapa de la parte derecha la ruta más corta a pie hasta la posición de la petición de favor, además de mostrar los datos relativos a la ruta, distancia y tiempo, en la información que se muestra.

Capítulo 9.

Gestión del proyecto

Este proyecto que ha desarrollado Faborez ha seguido la filosofía *Lean Startup* [25], que sin una planificación inicial establecida ha basado su desarrollo en una serie de principios a modo de *metaplanificación*. Estos principios generales han sido: el comienzo mediante un prototipo mínimo y funcional, la pronta introducción del *feedback* de los usuarios en el producto y la posterior ampliación progresiva del alcance en base a esta información.

Esta ampliación se ha producido mediante la interacción con los *txapeldun*es y la integración de sus aportaciones en el proyecto. De la misma forma, la gestión de la calidad se ha realizado con el seguimiento de sus comentarios y críticas, siendo la satisfacción de estos *tester*s el principal indicador.

Siguiendo la misma línea, los *txapeldun*es han sido considerados la parte interesada más importante del proyecto. Este proceso de comunicación, de recolección de *feedback*, ya se ha tratado más detalladamente en el capítulo 4.

El capítulo 3 también ha versado sobre los elementos tecnológicos que se han ido adquiriendo a lo largo de todo el proyecto, los ya han sido están debidamente detallados.

9.1. Gestión del alcance

Tal y como se ha mencionado anteriormente, el alcance del proyecto ha ido ampliándose a medida que este ha avanzado y como consecuencia del *feedback* de los *tester*s. El alcance del prototipo inicial, derivado de las historias de usuario planteadas en un inicio (ver apartado 2.1) era el siguiente:

- El registro de usuarios se realizará mediante un formulario de registro.
- Petición de favores, indicando un texto corto y otro descriptivo más largo.
- La visualización de las peticiones cercanas, y la posibilidad para responder a estas.
- Las respuestas se limitarán a respuestas simples, sin más conversación.

- La implementación se realizará solamente en formato *webapp*, sin incluir otros clientes.

Tras el prototipo, y con el *feedback* recibido de los primeros *txapeldun*es, se amplió el alcance con las siguientes características:

- Las peticiones se agruparán en categorías según su naturaleza. La lista se elaborará con las aportaciones de los usuarios.
- Todas estas peticiones de favor caducarán pasado un tiempo, que por el momento estará prefijado en la aplicación.
- Los usuarios podrán ver el perfil de otros usuarios, en el que se mostrará la actividad que estos tienen en Faborez.
- La respuesta a las peticiones derivará en un hilo de mensajes tipo *chat*.
- Se desarrollará una *app* para Android, que permita realizar todas las acciones nativamente, y que además reciba y muestre notificaciones sobre las peticiones cercanas. En el futuro serán solo los clientes móviles los únicos disponibles.

Después del segundo ciclo de recogida de comentarios se incluyeron en el alcance los siguientes puntos:

- Faborez dispondrá de un sistema de reporte de mal uso, para denunciar peticiones de favor con contenido inadecuado.
- Se añade el soporte multilingüe al cliente Android, con la traducción a Euskara.
- Las notificaciones serán configurables, pudiendo activar o desactivar tanto el sonido como la vibración por separado.
- Estos elementos de configuración se mostrarán en un apartado de ajustes dentro de la aplicación.

9.2. Gestión del tiempo

Los siguientes son los hitos más significativos que han tenido lugar en el desarrollo del proyecto:

5 de diciembre Comienzo de la implementación.

19 de diciembre Publicación del primer prototipo funcional y adquisición de primeras aportaciones.

21 de enero Lanzamiento de la segunda versión con las mejoras y comienzo de la implementación del cliente Android.

4 al 6 de febrero Lanzamiento del cliente móvil y entrevistas con los *txapeldun*es sobre el uso de este.

16 de febrero Alta de Faborez en Play Store.

1 de abril Recogida del último *feedback* de los usuarios.

25 de abril Lanzamiento de la última versión dentro del alcance del proyecto, y fin de las tareas de implementación.

5 de mayo Finalización de la primera versión completa de la memoria del proyecto.

12 de mayo Depósito de la versión final de la memoria.

9.3. Gestión de costes

Los costes económicos que el proyecto ha supuesto son despreciables respecto al volumen de trabajo, por lo que solamente se detallan, en horas, el coste humano que el proyecto ha supuesto (cuadro 9.1). Estas horas se dividen en dos categorías: por un lado, aquellas con un alto componente formativo y menos productivas por lo tanto, diferenciados mediante el texto «(for.)»; por otro, las actividades en las que ya se tenía experiencia, más productivas y que por tanto supondrían un mayor coste.

Categoría	**Tarea**	**Dedicación (h)**
Gestión del proyecto	Adquisición de tecnologías y servicios	20
	Reuniones de control	25
	Gestión del alcance	15
Backend	Diseño de la API	10
	Diseño y modificación del modelo de datos	30
	Implementación	30
	Implementación (for.)	55
	Implementación de servidor OAuth (for.)	25
	Corrección de errores	20
Cliente Android	Diseño gráfico de interfaz	20
	Lógica de la interfaz (for.)	60
	Capa de conexión al *backend* (for.)	20
	Corrección de errores	20
Aplicación web	Diseño de interfaces	5
	Implementación de lógica	10
	Correción de erores	5
Gestión de *feedback*	Motivación y promoción	15
	Preparación de cuestionarios	5
	Entrevistas y síntesis de los comentarios	12
Memoria	Redacción	45
	Aprendizaje de LaTeXy Maquetación	15

Cuadro 9.1.: Tabla de dedicación horaria

Capítulo 10.

Conclusiones

En este proyecto se ha logrado la puesta en marcha de Faborez, una aplicación social cuyas características han sido definidas por los propios usuarios mediante su participación continua en el proyecto. El implementación se ha realizado utilizando tecnologías en auge dentro del mundo del desarrollo web.

En un proyecto que plantea una arquitectura de estas características, y con vistas a facilitar la progresiva mejora, ha resultado vital hacer uso, en la mayor medida de lo posible, de protocolos diseños y librerías ya existentes tanto para descargar esta tarea como para asegurar la interoperatividad de los distintos componentes, como por ejemplo OAuth 2.0. Este protocolo es utilizado para la autenticación de peticiones por parte de clientes distintos al de la aplicación web, cuya especificación es pública. De hecho, se han utilizado librerías de distintos proveedores para cliente y servidor sin ningún problema de compatibilidad.

Se ha adquirido una infraestructura del modelo *Platform–as–a–Service* para alojar el *backend* de Faborez. Este tipo de servidores no requieren apenas de configuración y la instalación de la primera versión, así como la actualización a futuras versiones, solo requiere de unos poco minutos. Heroku ha sido el proveedor escogido, dado que disponía de los añadidos MongoDB y Redis, necesarios para el proyecto, y sin coste alguno. Proyectos de mayor escala requerirían un análisis más exhaustivo, ya que en el terreno de los servicios de pago las características son muy variadas.

El alcance del proyecto ha partido de un prototipo inicial, un *Minimum Viable Product*, implementado en forma de aplicación web, pero utilizando Sails.js para el *backend*, cuya flexibilidad permite adaptarlo fácilmente para distintos tipos de cliente.

Con este inicio, la dirección de las posteriores ampliaciones ha sido marcada por los *txapeldun*es, usuarios de la aplicación, con la visión y conocimiento suficientes para aportar propuestas no solo interesantes y creativas, sino viables en gran medida para su inclusión dentro del alcance del proyecto.

La participación de los *betatester*s no ha resultado ser trivial, ya que para optimizar la utilidad de su tiempo se han planificado las distintas entrevistas, acompañadas de cuestionarios que ayuden a centrar la atención en los aspectos que necesiten de *feedback*.

Así mismo, se ha procurado mantener la motivación de los usuarios con *feedback* sobre el progreso y pequeños alicientes. De hecho, la propia aplicación desarrollada ha resultado ser la vía idónea para «pedir el favor» de realizar las pruebas que en el momento fueran necesarias.

Además, la recogida de *feedback* no ha sido una tarea meramente organizativa. Se han utilizado herramientas tecnológicas que, sin conllevar trabajo al usuario, han recopilado los errores ocurridos durante la ejecución. Para esta tarea se ha utilizado el servicio Bugsense, desconocido previamente y que probablemente será utilizado en proyectos futuros fuera del ámbito de la universidad.

No obstante, se podría profundizar aún más en la búsqueda de herramientas tecnológicas para simplificar la gestión de comentarios, que en este caso se ha llevado artesanalmente. Los servicios de pago ofertan características como gestión de incidencias y recogida de *feedback* utilizando capturas de pantalla, por ejemplo, pudiendo los usuarios añadir este contenido *motu proprio*.

Esto último pasaría a ser un requisito indispensable si, en vez de forma individual, el desarrollo se hubiera llevado en equipo. En estas situaciones debe existir un sistema de información organizado que permita a cualquier miembro del equipo conocer el estado global del proyecto, junto con toda la información relevante, como son las aportaciones de los *txapeldun*es. En estos casos sería conveniente que hubiera un miembro del equipo dedicado a las tareas de interacción con los usuarios, intentando que estuviera menos influenciado por los asuntos técnicos. En la jerga utilizada en las metodologías ágiles a esta persona se le suele denominar «*product owner*».

No se debe olvidar que el fin último de cualquier ingeniero es poner la tecnología al servicio y utilidad de las personas. Para cumplir con ese fin no solo se debe conocer a fondo el área de estudio, sino que es indispensable saber tratar con las personas. Los proyectos que integran a usuarios reales desde la base, no solo son más enriquecedores y pedagógicos, sino que además logran romper con el estigma de que «los Proyectos Fin de Carrera se archivan y olvidan una vez dados por finalizados».

Capítulo 11.

Propuestas de mejora

Faborez ha sido un proyecto que desde su inicio ha ido evolucionando y al que se le han ido añadiendo distintas mejoras. No obstante, el proyecto, como trabajo académico, ha tenido que llegar a su final, y en este cierre se han tenido que dejar de lado muchas de las ideas de mejora que se han ido acumulando.

El listado a continuación reúne una relación de propuestas de mejora, que han sido recogidas y procesadas de entre todas las ideas aportadas por los *betatester*s, además de otras de origen propio que han ido surgiendo con el desarrollo, para que sirvan como propuesta para dar continuidad al trabajo presentado en esta memoria.

1. Un sistema por el cual el usuario pueda gestionar la caducidad de sus peticiones. Por ejemplo, se mostraría una notificación al usuario en su móvil informándole de la próxima caducidad de su petición. El usuario puede entonces reenviar la petición para intentar lograr alguna respuesta.

2. De la misma forma, puede que el usuario, después de hacer una petición, se mueva significativamente del lugar. En este caso también, la aplicación podría mostrar una notificación explicando la situación y ofreciendo la posibilidad de recolocar la petición. Igualmente, otra posibilidad sería eliminar la petición si esta ya hubiera perdido su validez con el movimiento.

3. En un escenario ideal, en el cual la gran mayoría de personas de una zona tuvieran instalado Faborez en sus móviles y realizaran peticiones regularmente, la cantidad de notificaciones que los usuarios pueden recibir puede ser inmenso, por lo que se requiere algún tipo de procedimiento para priorizar y filtrar las peticiones que se le aparecen a un usuario.

   Existirá un número máximo de peticiones que pueden mostrársele en un preciso momento. Si hubiera menos peticiones que este número se mostrarán todas, pero si hubiera más estas serían priorizadas en base a los siguientes criterios:

   a) Periodo de tiempo hasta la caducidad.

   b) Distancia desde el usuario hasta la petición.

   c) Relación con la persona que ha realizado la petición, si es un contacto conocido o no.

   d) Categoría de la petición.

   e) Karma del usuario que hace la petición.

   Estos criterios serían configurables por el usuario, pudiendo este ordenarlos según sus preferencias. Las categorías que se mencionaban también serían priorizadas de la misma forma.

4. Debido a la característica de cercanía, puede ocurrir que un usuario que no se encuentre lo suficientemente cerca de un núcleo significativo de usuarios perciba una sensación de soledad, que si la petición de otros se realizara a poca distancia no se daría.

   Por lo tanto, se ha propuesto como mejora un apartado que permita visualizar las zonas calientes en cuanto a peticiones de favores en la cercanía. Así el usuario sabe dónde se realizan más peticiones para realizarlas ahí, o para moverse y simplemente ayudar al resto de usuarios.

5. En la implementación actual, el cálculo de las distancias se realiza mediante una aproximación simplificada. La fórmula calcula que el usuario se encuentre en un rango de coordenadas de amplitud constante. Esta decisión se debe a haber premiado simplificar la función de búsqueda a la base de datos con miedo a que un procedimiento más exacto supusiera un perjuicio al rendimiento.

   Aunque esta aproximación funciona bien en la localización geográfica de Donostia y un error en la distancia no sería crítico, en latitudes muy distintas resultaría necesario revisar esta fórmula. Como se ha utilizado MongoDB como base de datos, este ya dispone de una función de búsqueda mediante coordenadas y solo se necesitaría analizar el modo de integrar esta búsqueda en el framework Sails.js.

6. Es posible que los dos campos de texto que Faborez ofrece para realizar la petición no sean suficientes para describir la petición de favor y que estos necesiten de acompañamiento contenido multimedia. Se propone por lo tanto que a las peticiones se les pueda adjuntar una fotografía, que acompañe a la explicación textual.

7. Una vez fuera realizada la integración entre Faborez y el servicio de Karma (ver apéndice C), los usuarios con mayor karma podrían tener una serie de ventajas a la hora de pedir favores. Pedirlas a una distancia mayor o que tuvieran una duración mayor, son algunos ejemplos.

8. Actualmente, los reportes de uso indebido tan solo se almacenan en la base de datos y el administrador debe acceder manualmente para revisarlos y actuar en consecuencia. Convendría que existiera un gestor de estos reportes, que fácilmente se pudiera visualizar la información relativa a la petición y una galería de acciones que se pueden realizar. Además, el los administradores deberían recibir notificaciones

automáticas cuando se produzcan estos reportes, ya sea mediante correo electrónico o incluso dentro de la propia aplicación.

9. Para no limitarse al *feedback* recibido por los medios habilitados en este proyecto, la aplicación podría incluir un apartado en el que enviar al desarrollador sugerencias, o reportar fallos que se hubieran encontrado.
10. En la actual arquitectura de la aplicación (ver capítulo 5 y apartado 7.1), todos los datos se almacenan de forma permanente únicamente en el *backend*. Esta es una decisión de diseño con implicaciones, por ejemplo, en cuanto a la gestión de datos de carácter personal. De cara al cumplimiento de la LOPD, convendría revisar si esta centralización es adecuada o algunos de los datos deben permanecer tan solo en los clientes.
11. Con la finalidad de que potenciales usuarios con alguna discapacidad pudieran usar Faborez, se deberían estudiar las formas más adecuadas de funcionamiento. Por ejemplo, las peticiones podrían realizarse mediante utilizando solamente la voz.

Bibliografía

[1] Adobe Systems Inc. *Phonegap*. URL: `http://phonegap.com/` (visitado 03-05-2014) (vid. pág. 38).

[2] The Apache Software Foundation. *Apache Cordova*. URL: `https://cordova.apache.org/` (visitado 03-05-2014) (vid. pág. 38).

[3] Matt Asay. «Why MongoDB? It's the developers, stupid». En: (11 de jun. de 2012). URL: `http://www.theregister.co.uk/2012/06/11/mongo_db/` (visitado 01-04-2014) (vid. pág. 28).

[4] DocumentCloud. *Backbone.js*. URL: `http://backbonejs.org/` (visitado 06-04-2014) (vid. pág. 39).

[5] Iñaki García García. «La Representación Estudiantil en la UPV/EHU en el periodo 2000-2007». En: *Movimientos estudiantiles. Resistir, imaginar, crear en la Universidad*. Ed. por Xabier Albizu Landa, Joseba Fernández González y Jon Bernart Zubiri Rey. Universidad del País Vasco/Euskal Herriko Unibertsitatea, 2008. Cap. 8, págs. 139-152. ISBN: 978-84-96993-05-1. URL: `http://dialnet.unirioja.es/descarga/articulo/2687231.pdf` (visitado 01-04-2014) (vid. pág. 3).

[6] Google Inc. *Chrome V8 Introduction*. URL: `https://developers.google.com/v8/intro` (visitado 01-04-2014) (vid. pág. 30).

[7] Google Inc. *Dashboards*. URL: `https://developer.android.com/intl/es/about/dashboards/index.html` (visitado 12-05-2014) (vid. pág. 25).

[8] Google Inc. *El API de rutas de Google*. 31 de ene. de 2013. URL: `https://developers.google.com/maps/documentation/directions/` (visitado 03-05-2014) (vid. pág. 76).

[9] Google Inc. *GCM Advanced Topics*. Cap. Messages with payload. URL: `https://developer.android.com/intl/es/google/gcm/adv.html#payload` (visitado 30-04-2014) (vid. pág. 54).

[10] Google Inc. *Google Maps V2. Getting started*. URL: `https://developers.google.com/maps/documentation/android/start#specify_permissions` (visitado 15-04-2014) (vid. pág. 97).

[11] Google Inc. *Google OAuth Client Library for Java*. URL: `https://code.google.com/p/google-oauth-java-client/` (visitado 30-04-2014) (vid. pág. 99).

[12] Google Inc. *Location APIs*. URL: `https://developer.android.com/intl/es/google/play-services/location.html` (visitado 29-04-2014) (vid. pág. 70).

[13] Google Inc. *Manifest.permission*. URL: https://developer.android.com/intl/es/reference/android/Manifest.permission.html (visitado 15-04-2014) (vid. pág. 95).

[14] Google Inc. *Storage Options*. URL: https://developer.android.com/intl/es/guide/topics/data/data-storage.html (visitado 29-04-2014) (vid. pág. 65).

[15] Evan Hahn. *Understanding Express.js*. 4 de mar. de 2014. URL: http://evanhahn.com/understanding-express/ (visitado 01-04-2014) (vid. pág. 30).

[16] Dick Hardt. *The OAuth 2.0 Authorization Framework*. Internet Engineering Task Force. 2012. URL: http://tools.ietf.org/html/rfc6749 (visitado 30-03-2014) (vid. págs. 28, 56, 60, 99, 104).

[17] M. Jones y Dick Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Internet Engineering Task Force. 2012. URL: http://tools.ietf.org/html/rfc6750 (visitado 30-04-2014) (vid. pág. 99).

[18] KnpLabs. *KnpBundles*. URL: http://knpbundles.com/ (visitado 01-04-2014) (vid. pág. 29).

[19] Sean Ludwig. «Why Platform–as–a–Service is poised for huge growth». En: (8 de oct. de 2012). URL: http://venturebeat.com/2012/10/08/paas-platform-as-a-service-explained/ (visitado 14-04-2014) (vid. pág. 42).

[20] MojoTech, LLC. *Marionette.js*. URL: http://marionettejs.com/ (visitado 06-04-2014) (vid. pág. 40).

[21] MongoDB Inc. *MongoDB Overview*. URL: http://www.mongodb.com/mongodb-overview (visitado 30-03-2014) (vid. pág. 28).

[22] Mikel Niño. «Cinco claves sobre entrevistas de "Customer Development" para que el cliente sea PARTE de la creación de tu startup». En: (24 de dic. de 2013). URL: http://www.mikelnino.com/2013/12/cinco-claves-entrevistas-Customer-Development-cliente-PARTE-de-la-creacion-de-tu-startup.html (visitado 07-04-2014) (vid. pág. 49).

[23] Mikel Niño. «Modelado y segmentación de clientes». En: (4 de abr. de 2014). URL: http://www.mikelnino.com/2014/04/visita-guiada-modelado-segmentacion-clientes.html (visitado 08-05-2014) (vid. pág. 47).

[24] Martin Odersky. *What is Scala?* URL: http://www.scala-lang.org/what-is-scala.html (visitado 30-03-2014) (vid. pág. 29).

[25] Enric Ries. *The Lean Startup. How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, 2011. ISBN: 978-0-307-88791-7. URL: http://theleanstartup.com/ (visitado 09-04-2014) (vid. pág. 77).

[26] Janessa Rivera y Rob van der Meulen. «Gartner Says Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013». En: (13 de feb. de 2014). URL: https://www.gartner.com/newsroom/id/2665715 (visitado 02-04-2014) (vid. pág. 38).

[27] Margaret Rouse. *What is agnostic?* URL: `http://whatis.techtarget.com/definition/agnostic` (visitado 08-05-2014) (vid. pág. 35).

[28] Sameer Singh. «Smartphone Market Share By Country — Q3 2013: Android Dominates Outside US, Windows Phone Grows In Europe». En: (18 de nov. de 2013). URL: `http://www.tech-thoughts.net/2013/11/smartphone-market-share-by-country-q3-2013.html` (visitado 02-04-2014) (vid. pág. 38).

[29] Twitter, Inc. *Bootrap. The most popular front-end framework for developing responsive, mobile first projects on the web.* URL: `http://getbootstrap.com/` (visitado 11-04-2014) (vid. pág. 41).

[30] Typesafe, Inc. *Play! Framework. The High Velocity Web Framework For Java and Scala.* URL: `http://www.playframework.com/` (visitado 08-05-2014) (vid. pág. 29).

[31] Ibai Valencia. «Análisis de soluciones innovadoras para desarrollar aplicaciones cliente–servidor con tratamiento avanzado de información en la nube». Tesis de máster. Universidad del Pais Vasco/Euskal Herriko Unibertsitatea, 10 de jul. de 2013. URL: `http://www.ibaivalencia.com/download/memoria-tfm-ibai-valencia/` (visitado 03-05-2014) (vid. págs. 28, 37).

[32] W3C Consortium. *Geolocation API Specification.* Ed. por Andrei Popescu. 24 de oct. de 2013. URL: `http://www.w3.org/TR/geolocation-API/` (visitado 03-05-2014) (vid. pág. 76).

[33] Dan York. «Node.js, Doctor's Offices and Fast Food Restaurants – Understanding Event-driven Programming». En: (25 de ene. de 2011). URL: `http://code.danyork.com/2011/01/25/node-js-doctors-offices-and-fast-food-restaurants-understanding-event-driven-programming/` (visitado 01-04-2014) (vid. pág. 30).

[34] ZURB, Inc. *Foundation. The most advanced responsive front-end framework in the world.* URL: `http://foundation.zurb.com/` (visitado 11-04-2014) (vid. pág. 41).

Glosario

Activity Clase de Android que implementa las interfaces de usuario y controla su ciclo de vida. 65, 67, 92

AJAX *Asynchronous JavaScript And XML*. 39

Amazon AWS Proveedor de servicios IaaS. Web: `http://aws.amazon.com`. 42, 43, 53, 92

Apache Servidor HTTP de código abierto. 42

API *Application Programming Interface*. 28, 35, 37, 39, 54, 59, 62, 65, 71, 76, 80, 102

app Aplicación, generalmente para dispositivos móviles. 19, 20, 36, 38, 78, 94

Backbone.js Librería de JavaScript de estructuración de aplicaciones web. 27, 39, 40, 54, 92

backend Parte interna del *software*, que procesa los datos según la lógica de la aplicación. 5, 7, 19, 20, 27, 41, 53, 54, 56, 57, 59, 60, 67, 70, 71, 80, 81, 85, 93, 99

Bootstrap Hoja de estilo CSS de licencia libre desarrollada por Twitter. 27, 41

bundle Módulo de Symfony, que da a este *framework* funcionalidad añadida. 29

bytecode Código intermedio más abstracto que el código de máquina, que se ejecutará en una máquina virtual. 36

cache Almacenamiento temporal de datos para la mejora del rendimiento. 31

Connect.js *framework* de *middlewares* para Node.js. 17, 31–33, 93

cookie Dato que un servidor web almacena en el navegador para su identificación en futuras peticiones. 31, 57

CSS *Cascading Style Sheets*. 41, 91

EJS *Embedded JavaScript*. Librería libre de plantillas para JavaScript. 35, 59

Express.js *microframework* basado en Node.js que implementa sobre éste la lógica de un servidor web. 17, 31, 32, 34, 35, 93

Foundation Hoja de estilos CSS desarrollada por ZURB. 41

Fragment Interfaz secundaria de Android, que puede ser reutilizada en varios *Activities*. 67

framework Estructura conceptual y tecnológica compuesta por herramientas, librerías, módulos, convenciones y una metodología de trabajo, que tiene como fin organizar y facilitar el desarrollo. 19, 27, 29–32, 34, 35, 37, 59, 91–93, 104, 105

GCM *Google Cloud Messaging*. 17, 20, 54–56, 59, 60, 70, 71, 96

Git *Software* de control de versiones distribuido, desarrollado por Linus Torvalds. 43, 62

Heroku Proveedor PaaS basado en Amazon AWS que da soporte a distintas plataformas de desarrollo. 20, 27, 41, 44, 53, 62, 64, 81

hook Método de interceptar llamadas a funciones o mensajes entre los componentes del *software* para alterar su funcionamiento de la forma deseada. 59

HTML *HyperText Markup Language*. 35–37, 39–41, 54, 59, 62, 92, 94

HTML5 Quinta versión del formato HTML, que incluye mejoras como la introducción directa de vídeos, audio y el acceso a la geolocalización mediante JavaScript, entre otros. 25, 36, 37, 76

HTTP *Hypertext Transfer Protocol*. 30–32, 54, 57, 91, 99

IaaS *Infraestructure–as–a–Service*. 15, 41, 42, 91

Jackson Librería de Java para la codificación y decodificación del formato JSON. 65

JSON *JavaScript Object Notation*. Formato de intercambio de datos alternativo a XML, basado en JavaScript. 17, 28, 35, 54, 55, 59, 65, 66, 92

LOPD Ley Orgánica de Protección de Datos de Carácter Personal. 26, 85

LSSI Ley de Servicios en la Sociedad de la Información. 26

Marionette.js Librería en JavaScript para ampliar la funcionalidad de Backbone.js tipos complejos de vista. 40, 54

Maven Herramienta de software para la gestión y construcción de proyectos. 45

microframework *Framework* relativamente ligero o de funcionalidades esenciales, comparado con tecnologías de la misma familia. 34, 35, 91

middleware Módulo de *software* intermediario entre dos o más elementos de la arquitectura. 17, 31, 32, 35, 59, 91

MongoDB Base de datos NoSQL de almacenamiento de documentos. 5, 7, 9, 20, 27, 28, 43, 53, 59, 60, 81, 84

MVC Modelo–Vista–Controlador. 39

MVP *Minimum Viable Product*. 5, 7, 9, 81

MySQL Sistema de gestión de bases de datos SQL, de código libre. 42

Node.js Plataforma de desarrollo de servidores basado en JavaScript. 17, 28, 30, 31, 34, 42, 43, 91, 93

NoSQL Base de datos no relacional, que no utiliza el lenguaje SQL. 20, 28, 92

PaaS *Platform–as–a–Service*. 15, 20, 41–43, 81, 92

Play! Framework *Framework* de desarrollo web basado en Scala. 29

POJO *Plain Old Java Object*. Objetos Java simples, sin más atributos o métodos que los esenciales para trabajar con sus atributos. 65

pull Modelo de comunicación cliente–servidor, en el cual son los clientes los que llevan la iniciativa y solicitan los datos al servidor. 54

push En una arquitectura cliente–servidor, el modelo en el cual el servidor por iniciativa propia envía un mensaje al cliente. 20, 37, 54, 56

Redis Motor de base de datos en memoria, que almacena los datos en forma de clave-valor. 5, 7, 9, 20, 43, 53, 59, 64, 81

REST *Relational State Transfer*. técnica de arquitectura *software* para el desarrollo de sistemas web distribuidos. 28, 39, 53, 54, 62, 71

SaaS *Software–as–a–Service*. 41, 43

Sails.js *Framework* de desarrollo de servicio web basado en Node.js, Connect.js y Express.js. 5, 7, 9, 13, 19, 27, 28, 30, 35, 59, 60, 62, 64, 81, 84

Scala Lenguaje de programación funcional y orientado a objetos. 29, 93

SQL *Structured Query Language*. 27, 28, 93

Symfony *Framework* para desarrollo web basado en PHP. 91

Telegram Aplicación de mensajería instantánea para móviles, alternativa a WhatsApp. 51

token Cadena de texto utilizada por un cliente de la aplicación para autenticarse ante el *backend*. 13, 45, 57, 58, 60, 62, 66, 99, 105

txapeldun Campeón, representante de los usuarios de Faborez. 19, 20, 47–51, 77–79, 81, 82

URL *Uniform Resource Locator*. 31, 40, 54, 59, 60, 62, 99

webapp *App* desarrollada mediante HTML y que, por tanto, es multiplataforma. 37, 53, 78

WebSockets Protocolo de comunicación bidireccional y *full-duplex*. 30

XMLHttpRequest Clase JavaScript para realizar peticiones HTTP asíncronas sin recargar completamente la página que se muestra. 39

Apéndice A.

Permisos de las aplicaciones Android

Se ha mencionado en la memoria que la incorporación de nuevas funcionalidades al cliente de Android ha supuesto el progresivo aumento, durante el desarrollo del proyecto, de permisos necesarios para ejecutar la aplicación. Este documento enumera los permisos utilizados y la función de cada uno dentro de Faborez.

Las siguientes líneas, extraídas desde el propio manifiesto (`AndroidManifest.xml`) de Faborez, son todos los permisos que se han utilizado.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission
    android:name="com.google.android.providers.gsf.permission.READ_GSERVICES" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.VIBRATE" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
<uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
<uses-permission android:name="android.permission.USE_CREDENTIALS" />
<uses-permission android:name="android.permission.READ_SYNC_SETTINGS" />
<uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.GET_TASKS" />
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
```

Extracto de código A.1: Permisos de Android utilizados en Faborez

A continuación se explica la utilidad de cada uno de estos permisos [13]. Estos se agrupan en función de la finalidad, por lo que se repetirán distintos permisos al solaparse las necesidades.

`INTERNET` y `ACCESS_NETWORK_STATE` Son los permisos necesarios para la conexión a Internet. El primero de ellos permite abrir cualquier tipo de conexión, y el segundo es útil para comprobar la conectividad del teléfono en un momento dado.

`ACCESS_COARSE_LOCATION` y `ACCESS_FINE_LOCATION` Son utilizados para la geolocalización, habilitando tanto el sistema basado en la red móvil como el GPS. Después en la implementación puede escogerse cuál de los dos sistemas utilizar según las necesidades de precisión y de consumo de energía.

`GET_ACCOUNTS`, `MANAGE_ACCOUNTS` y `AUTHENTICATE_ACCOUNTS` Habilitan a la aplicación crear sus propias cuentas de usuario y gestionarlas, de manera que estas se muestren en la lista de cuentas de usuario en el panel «Ajustes».

`READ_SYNC_SETTINGS` y `WRITE_SYNC_SETTINGS` Sumados a los anteriores abren la puerta a programar tareas de sincronización de datos automáticas, que son ejecutadas cuando el sistema operativo considera apropiado según parámetros internos.

`RECEIVE_BOOT_COMPLETED` Permiso para que la aplicación se ejecute al iniciar el sistema operativo. Esta característica es necesaria para activar el servicio en segundo plano que gestiona la localización de los terminales.

`com.google.android.c2dm.permission.RECEIVE` y `WAKE_LOCK` Permiten recibir notificaciones de GCM y gestionarlas. El segundo permiso es necesario para poder gestionar estas notificaciones aún cuando el dispositivo se encuentra en modo inactivo.

`VIBRATE` Como su nombre bien indica, permite añadir vibración a las notificaciones que la aplicación muestre.

`GET_TASKS` Otorga el permiso de visualizar la lista de aplicaciones que se encuentran ejecutando en un momento dado. En el caso de Faborez, permite a los servicios en segundo plano comprobar si es Faborez la aplicación actual y modificar su compotamiento en el caso de errores. Si Faborez está en pantalla se muestra la pantalla de error, y si no lo está se envía una notificación.

INTERNET, ACCESS_NETWORK_STATE, WRITE_EXTERNAL_STORAGE y com.google.android.providers-.gsf.permission.READ_GSERVICES En su conjunto, son los permisos mínimos para hacer funcionar Google Maps dentro de la aplicación [10]. Además de los permisos ya explicados, WRITE_EXTERNAL_STORAGE permite escribir a la memoria externa para guardar los mapas que se van mostrando y READ_GSERVICES permite acceder a los servicios de Google Play, necesario para el funcionamiento de Maps.

Apéndice B.

Reporte de error a Google

La autenticación de los clientes con el *backend* se realiza mediante claves de sesión en el caso de la aplicación web, y con el protocolo OAuth 2.0 en el cliente Android (ver apartado 5.3). Al estar este último protocolo estandarizado, suele ser conveniente utilizar librerías de terceros ya probadas con el fin de evitar errores.

Como el cliente Android es implementado en Java, la librería escogida fue la que Google pone a disposición del público, con el nombre *Google OAuth Client Library for Java* [11]. Esta librería contiene la clase `Credential`, que proporcionándole las URLs del llamado *token endpoint* [16, sec. 3.2] y los *token*s de acceso y refresco iniciales, automatiza el refresco del *token* de acceso cuando éste caduca, de forma transparente, tanto para el usuario como para el desarrollador.

Al utilizarlo, se comprobó que la librería no detectaba correctamente la caducidad del *token*, identificando el mensaje de caducidad del servidor como un error de autenticación desconocido, y obligando al usuario a realizar el proceso de *login* completo.

Tras un proceso intensivo de revisión del código de la librería y de *debugging*, se encontró el error en el código de la librería. Resulta que el protocolo OAuth 2.0 indica [17, cap. 3] que la invalidez de un *token* se notifica indicando entre las cabeceras HTTP el texto `error="invalid_token"`.

La librería de Google realiza esta detección mediante la siguiente expresión regular:

```
INVALID_TOKEN_ERROR = Pattern.compile("\\s*error\\s*=\\s*invalid_token");
```

Como se ve, esta expresión regular no tiene en cuenta las comillas del texto `invalid_token`, por lo que actua incorrectamente y no realiza el refresco del *token*. El código correcto debería ser el siguiente:

```
INVALID_TOKEN_ERROR = Pattern.compile("\\s*error\\s*=\\s*\"invalid_token\"");
```

Este error, junto con el código propuesto para arreglar, se envió al sistema de soporte de la propia librería[1], y actualmente, a la fecha de finalización de esta memoria, los desarrolladores de Google se encuentran investigándolo.

[1] https://code.google.com/p/google-oauth-java-client/issues/detail?id=88

Apéndice C.

Sistema federado de karma

Faborez es una aplicación social que se basa totalmente en la actividad y buena voluntad de las personas que la utilicen. Lo único que estas personas reciben es un reconocimiento por parte de la aplicación y en su caso de la persona a la que se ayudó.

Muchas aplicaciones ya existentes en la red gestionan este problema asignando una puntuación de karma a sus usuarios, de tal manera que unos pueden comprobar la buena voluntad de los demás. No obstante, esta puntuación nunca sale del contexto de cada aplicación, por lo que las personas que utilizan varios servicios no mantienen un karma acumulado, resultado de la federación de servicios que se reconocen mutua credibilidad.

Con la popularización de los servicios web, las distintas aplicaciones están cada vez más conectadas, delegando mucha lógica común a las aplicaciones sociales en servicios externos. En este contexto, se plantea el diseño de un sistema federado de karma, de tal manera que distintas aplicaciones web compartan una única puntuación de karma para cada uno de sus usuarios. A esta entidad compartida se le denominará «registro akásico».

C.1. Modelo de datos

Registro akásico

Es la representación karmática de una persona. Es única a través de varias aplicaciones, y contiene un valor de karma como resultado de sus interacciones en dichas aplicaciones. Sus datos:

- Identificador público, como referencia para que otras aplicaciones puedan referenciar el registro.
- Credenciales de acceso, en forma de usuario y contraseña, pero que podrán ser cambiados.
- Karma, en forma de valor numérico positivo o negativo.

Aplicación cliente

Los desarrolladores que deseen utilizar el servicio de Karma deberán registrar sus aplicaciones, tal y como se hace en otros servicios (como las redes sociales) que quieran utilizar sus APIs. Sus datos:

- Datos identificativos de la aplicación, como el nombre, descripción, logotipo, etc.
- Credenciales de autenticación, Utilizados por el servicio de karma para autenticar las peticiones que se le realizan.

Actos

Los *actos* son las acciones del registro akásico que derivan en un aumento o disminución de su karma, similares a las transacciones bancarias de un cliente de un banco. Estas transacciones podrán ser para uso privado, público, o incluso para ser visualizadas en una línea de tiempo de las acciones del usuario. Sus datos:

- Referencia del registro akásico y a la aplicación cliente.
- Variación del karma, en un número positivo o negativo.
- Tipo de acto, de entre los predefinidos por el servicio de karma, a saber: buena acción, mala acción, gasto de puntos, etc.
- Marca de tiempo.
- Referencia al acto en el interior de la aplicación cliente, en forma de URL (u otro aún no planteado), para que se pueda navegar a esta acción.
- Otros datos interesantes para el servicio de karma, como otros registros akásicos que han participado en la interacción.

C.2. API

C.2.1. Funciones del registo akásico

Crear registro El proceso de registro que crea el registro akásico de esta persona en el servicio de karma.

Enlazar aplicación cliente La persona da acceso mediante este procedimiento para que una aplicación cliente pueda realizar acciones en su nombre, que pueden ser en nivel de escritura o de lectura.

C.2.2. Funciones de aplicación cliente

Registrar aplicación cliente Este procedimiento dará de alta a la aplicación en el servicio de Karma, obteniendo sus credenciales de acceso y dando comienzo a su actividad.

Solicitar enlace Dará comienzo al proceso de enlace entre la referencia de usuario con su registro akásico en el servicio de Karma. Ver apartado sobre seguridad, apéndice C.4.

Obtener Karma Obtiene el valor de karma del registro akásico de un usuario de la aplicación cliente.

Registrar acto La aplicación cliente, a raíz de una acción que el usuario de su aplicación realiza, refleja este hecho en su registro akásico en forma de solicitud de aumento o reducción de la puntuación.

Obtener registros akásicos La aplicación cliente puede obtener por esta vía, de entre los registros que tiene asociadas a sus usuarios, aquellos que cumplan el criterio de karma especificado, dando unos valores mínimo y máximo.

C.2.3. Funciones de administrador

Establecer karma Establece para un registro dado un valor concreto de karma, sobreescribiendo cualquier otro anterior.

C.3. Lógica de funcionamiento interno

El servicio de karma, durante su ejecución y de forma transparente a las peticiones, tendrá implementadas los siguientes módulos de lógica.

C.3.1. Reglas de detección y prevención de fraude

Parece claro que en un servicio que tiene como resultado un valor de karma, será el interés de los usuarios obtener el mejor valor posible, lo que conlleva una amenaza de sudo de técnicas fraudulentas, y desvirtuando potencialmente tanto el valor que ofrece el servicio. La detección y prevención de estas técnicas se hará mediante un sistema de reglas, donde cada una de ellas pueda ser agregada o retirada de forma modular, siendo la ejecución del conjunto transparente para el resto de servicios.

Las reglas serán de dos tipos, según el momento en el que sean ejecutadas:

1. Las instantáneas o síncronas, las cuales se ejecutarán en el momento del registro de un acto, y podrán dar lugar a un bloqueo de este registro en el caso de una anomalía. Por su naturaleza síncrona, serán reglas de rápida ejecución.
2. Las periódicas o asíncronas, que serán ejecutadas en unos periodos de tiempo determinados o tras el registro de un número determinado de actos. Realizarán comprobaciones que afecten a un conjunto de actos y no a uno solo, pudiendo realizar operaciones más costosas en términos de computación.

Las consecuencias de que se incumplan esta serie de reglas podrá ser la de la eliminación de algún acto, penalizaciones en el karma del registro akásico o incluso el bloqueo del registro o la aplicación cliente que realice abusos graves.

C.3.2. Lógica interna

Asíncronamente al registro de los distintos actos de los registros akásicos, el servicio podrá modificar el karma de los usuarios, con el fin de que el funcionamiento y sus resultados se acerquen más a la realidad de lo que sería un karma.

Gestor de caducidad El servicio debe favorecer a los usuarios activos, de tal manera que el karma de los actos antiguos es «olvidado». Esto es, de forma periódica el karma será moderado por el propio servicio, de tal manera que tienda a cero a lo largo del tiempo si dicha persona no tuviera actividad.

C.4. Seguridad del servicio

En el servicio de karma entran en juego tres agentes en el momento de registro de un acto:

1. La persona que realiza el acto, que tiene en el servicio de karma un registro asociado.
2. La aplicación cliente, donde la persona ha llevado a cabo el citado acto, y que solicita al servicio el registro de este.
3. El propio servicio de karma, que recibe la solicitud y la registra.

El registro del acto no puede realizarse por cualquier aplicación cliente para cualquier registro akásico, sino que debe existir una autorización específica previa del registro para la aplicación concreta.

El diseño de los procedimientos que satisfacen esta necesidad ya se encuentra inventado y estandarizado en el *framework* OAuth (existiendo una versión número dos actualizada) [16].

Por lo tanto, el servicio solo debe implementar aquellas funcionalidades concretas contempladas en la especificación, a saber:

- El registro de aplicaciones cliente.
- Un caso de uso en el que se autorice a estos clientes a realizar acciones en nombre de un usuario.
- Los procedimientos que generen los respectivos *token*s de acceso para la aplicación cliente.

Por suerte, dada la popularidad de este *framework*, existen numerosas librerías prefabricadas, que cumplen los requerimientos básicos de seguridad, y que ahorrarían repetir esta funcionalidad con un esfuerzo asumible.