eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Facultad de Informática
Grado en Ingeniería Informática

▪ **P**royecto **F**in de **G**rado ▪

Ingeniería del Software

WeLive: desarrollo de aplicaciones móviles orientadas a *Smart Cities*.

Iñaki Latorre Ortega

Septiembre 2016

informatika fakultatea
facultad de informática

"The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time."

—Tom Cargill, Bell Labs

Resumen

Este documento corresponde a la memoria del proyecto de fin de grado "WeLive: desarrollo de aplicaciones móviles orientadas a *Smart Cities*", desarrollado para la titulación de Grado en Ingeniería Informática en la especialidad de Ingeniería del Software, en la Facultad de Informática de la Universidad UPV/EHU de Donostia-San Sebastián.

En este proyecto se han desarrollado, previa formación en distintas tecnologías, dos aplicaciones o *apps* móviles híbridas orientadas a *Smart Cities*. Además, la primera aplicación hace uso de dos servicios web que también forman parte del proyecto.

El proyecto ha sido realizado en la empresa Eurohelp Consulting S.L (en adelante Eurohelp), consultoría de referencia en Euskadi que ofrece soluciones innovadoras y eficaces en el ámbito de las TICs. En concreto, se enmarca en la sección I+D+i de dicha empresa bajo la supervisión de Leire Bardají y Aritz Rabadán en Donostia y de Alex Novoa en Bilbao, y dirigido por el profesor de la UPV/EHU Óscar Díaz.

Por último, cabe destacar que el proyecto fin de grado forma parte de otro mayor a nivel europeo, llamado WeLive, en el que participan distintas empresas u organizaciones. Ha habido que colaborar y tratar con algunas de ellas como Tecnalia o la Universidad de Deusto en España, o Foundazione Bruno Kessler (en adelante FBK) en Italia. Dicho proyecto tiene como ciudades piloto Bilbao en España, Trento en Italia, Novi Sad en Serbia y la región de Helsinki en Finlandia. En este caso las aplicaciones desarrolladas son para la ciudad piloto Bilbao.

Índice

Lista de figuras y tablas

FIGURAS

TABLAS

Capítulo 1

1. INTRODUCCIÓN

En este capítulo se comenta el contexto en el que se ha desarrollado el proyecto fin de grado, tanto por parte de la empresa como por parte del proyecto WeLive a nivel europeo en el que se enmarca.

Por otro lado, se realiza una descripción general del presente documento con objeto de introducir al lector las secciones que se van a detallar posteriormente.

Este documento corresponde a la memoria del proyecto realizado por el alumno del Grado en Ingeniería Informática Iñaki Latorre Ortega, en la sede de Donostia-San Sebastián de la empresa Eurohelp. El trabajo se ha desarrollado de manera autónoma bajo la supervisión de Leire Bardají en relación a la planificación del mismo, y de Aritz Rabadán en relación a aspectos técnicos y de colaboración con otras empresas. Puntualmente también se ha colaborado con Alex Novoa, diseñador gráfico afincado en la sede de Bilbao de la mencionada empresa.

El propósito de este proyecto es el de obtener dos aplicaciones orientadas a la estrategia de *Smart City* que Bilbao tiene, haciendo uso de diferentes componentes y librerías que se irán comentando en los apartados siguientes. La primera, además, hace uso de dos servicios web que también se han desarrollado. Por tanto, es cuatro (4) el número de artefactos software que se han conseguido en este trabajo fin de grado.

Tal y como se va a comentar en el siguiente capítulo (2. Antecedentes y estado del arte) las aplicaciones realizadas para la ciudad piloto de Bilbao están incluidas en el proyecto WeLive y, por tanto, hacen uso de componentes desarrollados en la plataforma WeLive. Dichos componentes han sido implementados en parte en la empresa Tecnalia y la Universidad de Deusto, y en consecuencia ha habido una estrecha colaboración con dichas empresas. En concreto con el Dr. Iñaki Maestro y el Dr. Unai Aguilera respectivamente, en relación a la integración y uso de los mismos en las aplicaciones.

En Tecnalia también se han desarrollado otras dos aplicaciones para Bilbao que deben tener un estilo gráfico similar a las desarrolladas en este proyecto fin de grado, y por tanto también ha habido una comunicación con el Dr. Iñaki Maestro en este sentido. Además, puntualmente y junto con Aritz Rabadán, se han mantenido contactos con Raman Kazhamiakin de la empresa FBK de Italia para el despliegue de los servicios web desarrollados.

El presente documento se estructura en los siguientes capítulos:

- El documento de antecedentes del proyecto, en el que se menciona la experiencia previa en Eurohelp, el proyecto europeo WeLive y el estado del arte en cuanto al desarrollo de las aplicaciones móviles.
- El documento de objetivos del proyecto, con el alcance inicial y las modificaciones que han ido surgiendo en él, y las exclusiones.
- El documento que recoge la metodología de trabajo seguida y las herramientas utilizadas para ello en la empresa Eurohelp.
- Repaso a las herramientas y tecnologías utilizadas en el desarrollo de los artefactos software, incidiendo en las que se consideran relevantes y de valor añadido debido a la inexperiencia en las mismas y/o su novedad.
- El documento que recoge el Ciclo de Vida de cada uno de los cuatro artefactos desarrollados (dos servicios web y dos aplicaciones).
- El documento que recoge la gestión del proyecto, incluyendo la planificación inicial con la formación y desarrollo, el aumento en el alcance del mismo que se ha producido por las exigencias de los encargados de las aplicaciones de WeLive a lo largo del proyecto, o la comparativa final entre lo estimado y lo real.

- El documento que recoge las conclusiones tanto a nivel de objetivos y de gestión como a nivel personal.
- El documento en el que se describen brevemente propuestas de mejora del trabajo realizado.

También se añade un apartado de referencias bibliográficas, además de otro en relación al glosario y acrónimos que al lector le puede resultar útil para la correcta comprensión de algunos términos o siglas. Además, se adjuntan al presente documento diferentes anexos a los que se hacen referencia en los apartados correspondientes.

Capítulo 2

2. ANTECEDENTES Y ESTADO DEL ARTE

En este capítulo se realiza un repaso a los antecedentes del proyecto fin de grado. Se describe el contexto general del trabajo en Eurohelp, del proyecto WeLive y del estado del arte en relación al desarrollo de aplicaciones móviles nativas vs híbridas.

2.1. Prácticas en Eurohelp

Mi comienzo de la relación laboral con Eurohelp se remonta al verano de 2015, una vez terminado el curso 3º del Grado en Ingeniería Informática. El convenio de prácticas se produjo con vistas a integrarme en la empresa, conocer un entorno laboral profesional y tener la posibilidad de realizar el proyecto fin de grado en la empresa, como así ha sido.

El primer periodo de prácticas realizado hasta el inicio del 4º curso en septiembre de 2015 supuso una buena relación con la empresa tanto a nivel personal como por el trabajo realizado. Me enmarcaron en la sección de I+D+i con un proyecto que consistía en la elaboración de una página web responsiva utilizando Spring, con una parte servidora con Spring MVC y utilizando JSP para la parte web. Por tanto, tuve que formarme en algunas tecnologías poco o nada trabajadas durante la carrera, algunas de las cuales me han servido en el desarrollo del proyecto fin de grado como el uso de la API para JavaScript de Google Maps o Spring MVC.

Además, la novedad consistió en que el proyecto se enmarcaba en la web semántica, para lo que tuve que formarme en esa filosofía y trabajar con una base de datos no conocida hasta el momento que trabajaba con *Linked Data*, teniendo que aprender la forma de realizar consultas SPARQL, entre otros, siendo el origen de datos el ofrecido por diferentes entidades públicas de Guipúzcoa (http://www.gipuzkoairekia.eus/es).

Para no perder la continuidad en la empresa, a partir de septiembre de 2015 continué con otro periodo de prácticas en la empresa compaginándolo con el primer cuatrimestre del 4º curso del grado. Esto me ha servido para que me ofrecieran desarrollar el proyecto fin de grado a partir de enero de 2016, especializándome en cierto modo en el desarrollo de aplicaciones móviles híbridas y conociendo más las relaciones laborales con compañeros de esta y otras empresas. A fecha de entrega del presente documento me encuentro ya trabajando en Eurohelp.

2.2. Proyecto WeLive

Tal y como se ha comentado en la introducción (capítulo 1), el proyecto fin de grado realizado forma parte de otro proyecto más grande a nivel europeo llamado WeLive. En él forman parte diferentes empresas, organizaciones y Administraciones Públicas (en adelante A.P.) de España (Eurohelp, Universidad de Deusto, Tecnalia y el Ayto. de Bilbao), de Italia (FBK o Comune di Trento, entre otros), de Serbia (por ejemplo, Company Informatika Novi Sad) y de Finlandia (Cloud'N'Sci Ltd). Las ciudades piloto elegidas para los servicios desarrollados son Bilbao en España, Trento en Italia, Novi Sad en Serbia y la región de Helsinki en Finlandia.

WeLive es un proyecto internacional cuyo objetivo principal es el de reducir la brecha entre la innovación y la adopción de los servicios del *Open Government* y el de impulsar los servicios públicos implicando a ciudadanos y empresas en el diseño, creación, selección y entrega de algunos servicios públicos en forma de aplicaciones móviles. Para lograr el objetivo, las compañías desarrollan los llamados *Building Blocks* (en adelante, BB) que cumplen las ideas seleccionadas por los ciudadanos, A.P. y compañías.

Por tanto, WeLive proveerá un nuevo ecosistema (plataforma) de herramientas de *e-Government* construidos con los paradigmas *Open Data*, *Open Services* y *Open Innovation*, que permitan implicar a los ciudadanos, A.P. y empresas en el proceso de innovación de la ciudad o territorio correspondiente. De este modo, WeLive ofrece los siguientes puntos:

- Que los ciudadanos, compañías y A.P. expresen sus necesidades y propongan nuevas ideas para servicios y apps.
- Que las compañías desarrollen *Building Blocks* a partir de las necesidades identificadas.
- Que las A.P. y los ciudadanos (usuarios finales) ofrezcan datos en forma de *Open Data* que surgen de las necesidades expresadas.
- Que las A.P. y los ciudadanos financien las compañías para transformar ideas populares en aplicaciones, utilizando BB's (*Open Service* y *Open Data*) disponibles en el *marketplace* de la plataforma WeLive.

Figura 1: Objetivos tecnológicos y sociales de WeLive [1]

[1] Fuente: Documentación oficial de WeLive, sección 1-3

La Figura 1 ilustra los objetivos principales del proyecto WeLive que se han mencionado en general, tanto desde el punto de vista social como tecnológico. Por ejemplo, se quiere promover el crecimiento económico y la creación de empleos a través de una serie de herramientas para crear y desplegar nuevos servicios públicos (apps) a partir de *datasets* abiertos o libres, o adaptar nuevos *datasets* a través del componente *Open Data Toolset* de WeLive para dar acceso público a nueva información de interés público.

En el aspecto tecnológico, entre otros, está el objetivo de democratizar la creación de nuevos servicios públicos. Es decir, se ofrece una plataforma de colaboración abierta para A.P. donde los servicios básicos entorno a los datos que pertenecen a ayuntamientos e incluso los servicios básicos que acceden a datos externos al ayuntamiento puedan ser combinados por cualquier parte, desde el personal de la A.P, pasando por los desarrolladores de software hasta poner fin en los ciudadanos finales.

Otro objetivo en relación a la tecnología de WeLive corresponde a la explotación eficiente de los datos abiertos (*Open Data*) en los servicios públicos. Se quiere hacer posible el acceso a dichos datos procedentes de diferentes fuentes heterogéneas como repositorios de datos *Open Government* o datos suministrados por el usuario a través de redes sociales o aplicaciones, entre otros. Para ello se ofrecen los componentes *Open Data Toolset* (conjunto de herramientas de *Open Data*) antes mencionado y el *Citizen Data Vault*.

Concretando el ámbito del trabajo de fin de grado

La plataforma y todo lo que la envuelve de WeLive es muy extensa y solo se ha comentado el contexto general del proyecto, ya que no hace falta entrar al detalle. Sin embargo, de cara a analizar el ámbito en el que se ha desarrollado el proyecto de fin de grado dentro de WeLive conviene aclarar algunas cosas:[2]

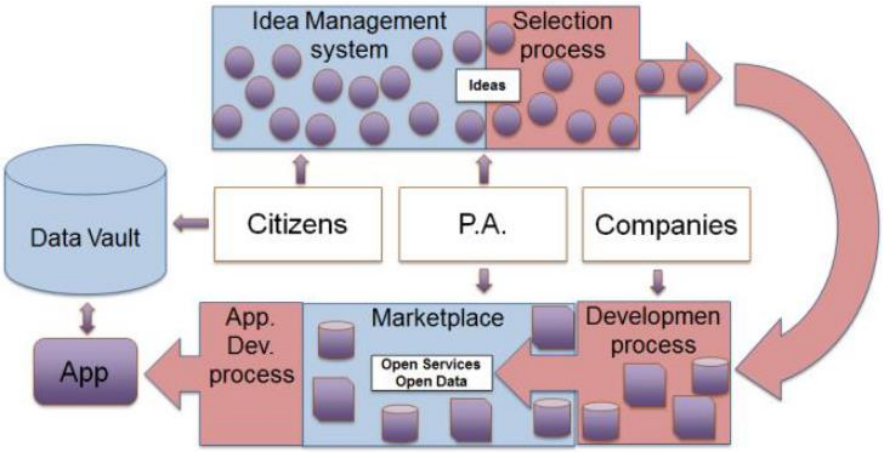


Figura 2: Cadena de montaje general de WeLive [2]

[2] Fuente: Documentación oficial de WeLive, sección 1.3

En primer lugar, la idea principal detrás del concepto de WeLive es la de una cadena de montaje (ver la Figura 2 de la página anterior). En dicha imagen los componentes se muestran en azul, los procesos en rojo y los *ítems* (ideas, *Building Blocks*, apps) en morado. La infraestructura de *Open Government* propuesta ofrece herramientas para transformar necesidades en ideas, a continuación herramientas para seleccionar las mejores ideas y crear los *Building Blocks* necesarios para construir las soluciones previstas, y finalmente una manera de componer estos BB en aplicaciones que pueden ser explotadas a través del *marketplace* de WeLive. En pocas palabras, la colaboración público-privada de los grupos de interés da lugar a las ideas convertidas en aplicaciones y explotadas en el mercado de WeLive.

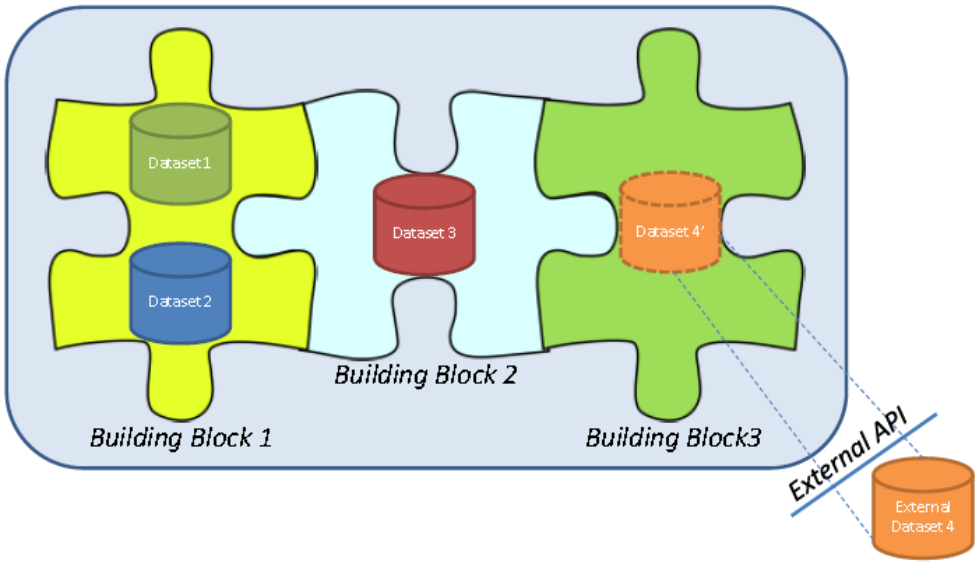


Figura 3: Composición de un servicio público de WeLive (app) [3]

Y en segundo lugar, en la Figura 3 se puede apreciar la composición de un servicio público tal y como lo entiende WeLive, en la forma de una aplicación móvil con un diseño de interfaz responsivo (RWD o *Responsive Web Design*) ejecutable en diversos dispositivos móviles por los usuarios finales, normalmente ciudadanos. Una vez desarrollada es desplegada en el Entorno de Ejecución de WeLive para el Ayto. correspondiente, y publicada a través de su *marketplace*.

En dicha ilustración se muestran los *Building Blocks* de los que una app se compone. Un BB normalmente ofrece acceso a una serie de *datasets* o bases de datos, y aísla a los desarrolladores de aplicaciones de tener que acceder a *datasets* concretos que pueden estar en formato RDF (*Resource Description Framework*, propios de la web semántica), datos de redes sociales o incluso datos del sector privado. Cada bloque (BB) aplica la lógica de negocio necesaria alrededor de los *datasets* a los que accede y ofrece una REST API que puede ser explotada durante la composición de las aplicaciones.

[3] Fuente: Documentación oficial de WeLive, sección 1-3)

Una vez explicado esto, el trabajo de fin de grado forma parte tanto de los *Building Blocks* como de los servicios públicos (apps) mencionados. Si bien esto se analizará más en profundidad en el capítulo 6, se han realizado dos aplicaciones y dos *Building Blocks* (utilizados por la primera aplicación). Las dos aplicaciones hacen uso también de otros BB's o componentes de WeLive (de ahí la colaboración con Tecnalia o Deusto), como el sistema de autenticación de un usuario en las apps o un sistema de *logging* interno de las apps de cara a registrar algunas actividades para los administradores de WeLive.

En primeras fases del proyecto, a inicios de 2015, se definieron una serie de servicios públicos para cada ciudad piloto del proyecto. En este caso, para Bilbao, se definieron siete servicios distintos junto con los BB's que cada uno debe usar, dos de los cuales han sido realizados en Tecnalia y otros dos son los realizados en el proyecto fin de grado: "Bilbozkatu" y "BilbOn". El primero consiste en la gestión de propuestas por parte de los ciudadanos de cara a mejorar la ciudad (pudiendo votarlas, etc.), y el segundo sirve para encontrar puntos de interés (obtenidos de datos proporcionados por el Ayto. de Bilbao) de la ciudad además de que los ciudadanos creen los suyos.

Más información sobre el proyecto WeLive en http://www.welive.eu/.

2.3. Estado del arte

El mercado de las aplicaciones móviles ha crecido mucho los últimos años, y con ello, la forma de desarrollarlos. Por lo tanto conviene analizar las diferentes opciones que hoy en día existen al crear una app: nativa, web o híbrida.

NOTA
Se denomina **desarrollo nativo** al uso de las tecnologías propias de cada plataforma para el desarrollo de una aplicación. Es decir, lenguaje Java para plataforma Android y Objective-C o Swift para plataforma iOS. Requiere **tantos desarrollos como plataformas** se quieran soportar. Se denomina **desarrollo web** al uso de tecnologías web estándares, como HTML5, CSS y JavaScript, habitualmente utilizando técnicas de RWD para su visualización en múltiples plataformas. Sin embargo, ofrece una **peor experiencia de uso** e **ignora las características del dispositivo**. Se denomina **aplicación híbrida** a las aplicaciones desarrolladas con tecnologías web (HTML5, CSS3 y JavaScript) y que son **posteriormente empaquetadas como aplicaciones móviles instalables en dispositivos móviles de multitud de plataformas**.

Tabla 1: Definiciones de tipos de desarrollo de apps

La elección del tipo de desarrollo influirá en diferentes factores como son la tecnología de desarrollo, el rendimiento o el coste. En este caso ha sido un requisito de la empresa que el desarrollo de las aplicaciones haya sido de forma híbrida, utilizando para ello el Framework Ionic que se analizará más en profundidad en la sección 5.1. Sin embargo, a continuación se muestran los pros y los contras de cada tipo de desarrollo.

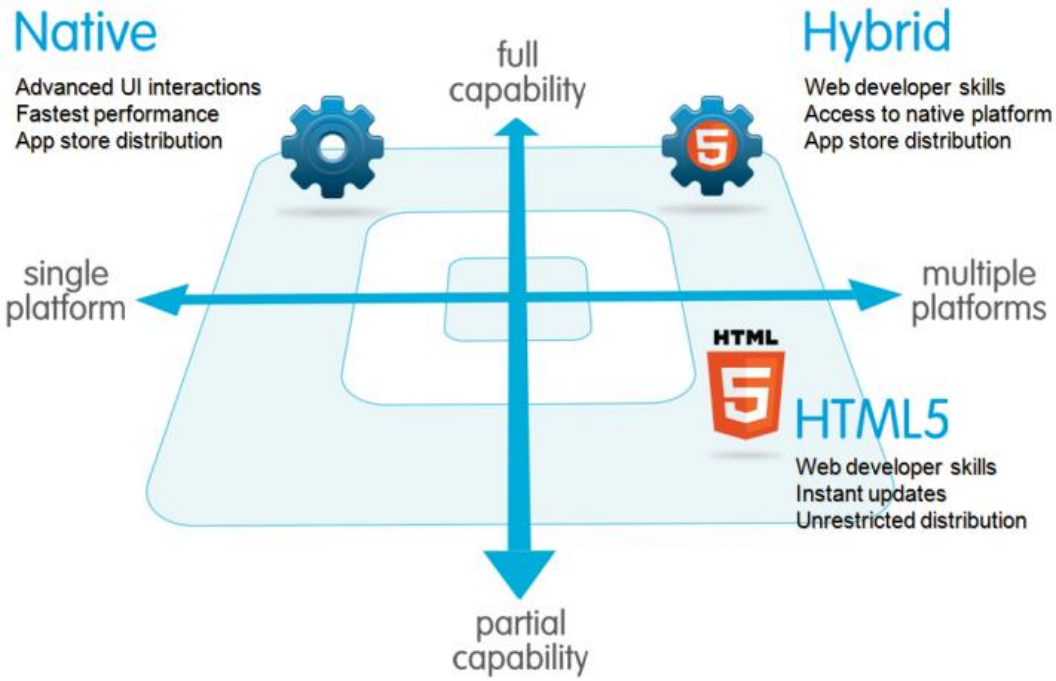


Figura 4: Gráfico ilustrativo de tipos de desarrollo de apps [4]

En la figura 4 se pueden ver gráficamente algunas ventajas y desventajas de los tres tipos de desarrollo de aplicaciones que se han mencionado. Principalmente, el desarrollo nativo ofrece un gran rendimiento a cambio de un gran coste y tiempo de desarrollo. El desarrollo web, en cambio, tiene la gran desventaja del rendimiento aunque el coste sea mucho menor.

Finalmente, el desarrollo de aplicaciones de forma híbrida aúna en gran medida las ventajas de los otros dos tipos, y minimiza los puntos negativos de los mismos. Así, la utilización de tecnologías web hace que la disponibilidad de los desarrolladores aumenta debido a que son tecnologías ampliamente conocidas (no hacen falta conocimientos específicos de cada SO en el que se quiera utilizar la app), además de que los desarrollos sean mucho más rápidos y de más fácil mantenimiento.

A continuación se muestran otras ventajas del desarrollo híbrido de apps:

- **Reutilización de un mismo código fuente**. Otra de las principales ventajas es el hecho de poder reutilizar el mismo código de la aplicación

[4] Fuente: http://i2.wp.com/blog.aplicacionesmovil.com/wp-content/uploads/Native_html5_hybrid.png

para varias plataformas. Esto reduce significativamente los costes de desarrollo y de mantenimiento posterior. La única tarea específica por plataforma a realizar es el empaquetado exterior que envuelve la aplicación, pero el código de ésta (HTML5, CSS3 y JavaScript) sigue siendo el mismo.

- **Acceso a funcionalidades nativas mediante plugins**. Diversos *frameworks* de desarrollo proveen de mecanismos para acceder a funcionalidades nativas de los dispositivos móviles como cámara y galería de fotos, agenda de contactos, GPS, acelerómetro… Por lo que es posible conseguir un rendimiento eficiente incluso cuando se hace uso de recursos del sistema.
- **Déficit de rendimiento comparado con desarrollos nativos en constante disminución**. Tanto la optimización de los *frameworks* de desarrollo como las capacidades de los dispositivos móviles soportadas por hardware están en evolución exponencial, hacen que la principal desventaja de las soluciones híbridas, sea cada vez menos significativa.

La principal desventaja de los desarrollos híbridos frente a los nativos, como se comentaba en el último punto, es que el rendimiento suele ser algo inferior en comparación con desarrollos nativos. Esta disminución de rendimiento se aprecia, principalmente, cuando las aplicaciones hacen uso intensivo de recursos hardware del sistema (brújula, GPS, acelerómetro, cámara, etc.). No obstante, cuando las aplicaciones no requieren utilizar estos elementos, o bien el uso es muy puntual (por ejemplo, obtener una coordenada GPS), la disminución de rendimiento apenas es apreciable. Además, la evolución tecnológica de los *frameworks* para desarrollos híbridos está provocando que esta diferencia sea cada vez menor y, dependiendo del tipo de aplicación, puede llegar a ser inapreciable.

Hay que valorar siempre el tipo de aplicación que se pretende desarrollar, pues las funcionalidades a soportar van a marcar la necesidad o no de un desarrollo nativo. Por ejemplo, si se trata de una aplicación que va a hacer un uso intensivo de funcionalidades gráficas (como juegos) o unos requisitos de computación grandes, bien por el tamaño de la aplicación o por los procesamientos que ella requiera, la solución debe siempre orientarse al desarrollo de una aplicación nativa.

Si por el contrario se trata de una aplicación con formularios, textos e imágenes y que no requiere de un uso intensivo de la tarjeta gráfica, **un desarrollo híbrido aporta todas las ventajas mencionadas previamente**. Como ya se ha comentado, en este proyecto fin de grado las aplicaciones se han desarrollado de manera híbrida con el Framework de Ionic, que está construido sobre la capa proporcionada por Apache Cordova para el acceso a funcionalidades del dispositivo, como se analizará en el apartado 5.1.

En la siguiente tabla (número 2) se resumen las principales ventajas e inconvenientes de los tres tipos de desarrollo de aplicaciones comentados:

	NATIVO	HÍBRIDO	WEB
Lenguaje	Java, -C, .NET	HTML5, CSS3, JavaScript	HTML5, CSS, JavaScript
Coste desarrollo	✗	⚠	✓
Tiempo desarrollo	✗	⚠	✓
Rendimiento	✓	⚠	✗
Multiplataforma	✗	✓	✓
Interfaz usuario	✓	✓	⚠
App Stores	✓	✓	⚠

Tabla 2: Comparativa de desarrollo de apps nativo, híbrido y web [5]

Respecto a la tabla 2, tal y como se ha comentado, se debe recordar que los puntos marcados como intermedios (icono amarillo) en relación al desarrollo híbrido indican que siempre se deben valorar las características del dispositivo que la aplicación vaya a necesitar. En el proyecto fin de grado, por ejemplo, no se hace un uso exhaustivo de dichas características, tan solo del GPS en una de las apps y del uso de un navegador interno de la aplicación o *inAppBrowser*, por lo que no supone un gran problema con respecto al rendimiento o coste de desarrollo.

Como último apunte en relación a las *App Stores* o tiendas de aplicaciones, decir que una aplicación web no puede ser gestionada por ninguna de estas tiendas para poder buscarla, etc., a diferencia de las aplicaciones nativas e híbridas que pueden empaquetarse para las distintas plataformas como Android e iOS. Además, las aplicaciones web requieren siempre de acceso a internet, cosa que no sucede necesariamente para los otros tipos de aplicaciones (a no ser que lo necesiten para acceder a servicios externos).

[5] Fuente: http://www.raona.com/es/Solutions/Template/163/App-nativa-web-o-h%C3%ADbrida-

Capítulo 3

3. OBJETIVOS

El proyecto de fin de grado se ha enmarcado dentro del proyecto WeLive presentado en el punto 2.2. En concreto, se deben realizar dos aplicaciones móviles de manera híbrida haciendo uso de componentes de WeLive, y de otros dos componentes que también se deben desarrollar para una de las mismas, orientadas a la estrategia de *Smart Cities* de Bilbao.

A continuación se describe el alcance del proyecto y las exclusiones que se definieron en la planificación del mismo.

3.1. Alcance del proyecto

El alcance definido en la planificación inicial se ha visto alterado en el transcurso del proyecto debido a diferentes razones. Esto ha supuesto un impacto en la dedicación y fechas estimadas para el proyecto fin de grado que en el capítulo 7 de Gestión del Proyecto se detallará. Se podrían haber excluido del proyecto los imprevistos que ha habido y que han provocado esta alteración, teniendo como resultado aplicaciones con menos funcionalidad y sin estar desplegadas públicamente. Sin embargo, dicho impacto se ha asumido de cara al proyecto fin de grado ya que los problemas tenían que ser solucionados obligatoriamente en el proyecto WeLive y existía la posibilidad de retrasar la entrega del proyecto fin de grado a una convocatoria posterior. Por tanto, el alcance del proyecto fin de grado finalmente ha sido el mismo al del proyecto WeLive en Eurohelp.

En general, el alcance consiste en el desarrollo de dos aplicaciones móviles híbridas para su uso en Bilbao. A continuación se muestran los puntos definidos como objetivos, a grandes rasgos, del proyecto (los marcados con un (*) indican la misma funcionalidad en las dos apps):

1. App BilbOn

    1.1. Consiste en una aplicación híbrida para la gestión de puntos de interés (en adelante, POIs o *Points of Interest*) de la ciudad de Bilbao. Debe estar disponible en Google Play y accesible, previa instalación en el dispositivo, a través del icono correspondiente.

    1.2. (*) Inicialmente se muestra un *splash screen* con los efectos, logotipos, y textos correspondientes a WeLive, el Ayto. de Bilbao y la Comisión Europea.

    1.3. (*) La primera vez se muestran los Términos y Condiciones de uso de la aplicación, que deben ser aceptados para acceder a la app.

    1.4. (*) Tiene soporte multilenguaje: castellano, euskara e inglés.

    1.5. Dispone de un mapa en el que visualizar los POIs sobre la ciudad.

    1.6. El usuario puede, a través del menú, filtrar los POIs en función de distintos criterios: categoría, ubicación (GPS o seleccionada a través de la herramienta *Google Places* de *Google Maps*), texto o una combinación de ellos.

    1.7. El usuario puede seleccionar un POI del mapa y visualizar sus detalles.

1.8. Si está autenticado, el usuario (normalmente, ciudadano) puede crear un nuevo punto de interés especificando distintos parámetros como el título, la descripción o la localización.

1.9. Se hace uso del **componente ODS (*Open Data Stack*) de WeLive** para la consulta de POIs ofrecidos por el Ayto. de Bilbao (datos abiertos) y la creación de nuevos por parte de los ciudadanos. Este componente aborda los desafíos de acceder al conocimiento de una ciudad en forma de múltiples orígenes de datos en formatos diversos.

1.10. (*) El usuario puede autenticarse a través de un botón en la pantalla principal: se hace uso del **componente AAC (*Authentication and Authorization Control System BB*) de WeLive** para la autenticación de usuarios.

1.11. (*) Dispone de una pantalla *About* en la que se muestra información de la aplicación y desde la que el usuario puede ejecutar un cuestionario sobre la app hasta cuatro veces (gracias al uso de un servicio web externo desarrollado en Tecnalia).

1.12. (*) Se hace uso del **componente *Logging BB* de WeLive** para registrar ciertos eventos en relación a la actividad de la aplicación, por ejemplo, "app iniciada" o "POI creado por el usuario x". Para ello la aplicación debe autenticarse también ante la plataforma WeLive (distinto a la autenticación del usuario, pero a través del mismo componente AAC BB).

2. App Bilbozkatu

2.1. Consiste en una aplicación híbrida para la gestión de propuestas de ciudadanos para mejorar la ciudad de Bilbao. Debe estar disponible en Google Play y accesible, previa instalación en el dispositivo, a través del icono correspondiente.

2.2. Una propuesta consiste en una sugerencia por parte de un usuario y dirigida al resto de ciudadanos y finalmente al Ayto., en la que se indica el título, la zona o barrio a la que se refiere, la categoría y la descripción. Al cabo de un tiempo la propuesta expira y pasa su estado de abierto a cerrado, impidiendo que los ciudadanos puedan votarla o enviar más *feedbacks* sobre ella.

2.3. (*) Inicialmente se muestra un *splash screen* con los efectos, logotipos, y textos correspondientes a WeLive, el Ayto. de Bilbao y la Comisión Europea.

2.4. (*) La primera vez se muestran los Términos y Condiciones de uso de la aplicación, que deben ser aceptados para acceder a la app.

2.5. (*) Tiene soporte multilenguaje: castellano, euskera e inglés.

2.6. Dispone de una pantalla inicial en la que poder buscar y visualizar las propuestas creadas por los ciudadanos en forma de lista (indicando sólo algunos datos). Se pueden ordenar en función de algunos parámetros (más nuevas primero, mejor valoradas...) y filtrar en función de la zona, categoría, texto o una combinación de ellos.

2.7. El usuario puede seleccionar una propuesta de la lista para ver sus detalles y acceder a nuevas funcionalidades:

2.7.1. Si el usuario está autenticado y la propuesta en estado "abierta", votar a favor o en contra de la propuesta. Sólo se admite un voto por propuesta y usuario.

2.7.2. Si el usuario está autenticado y la propuesta en estado "abierta", enviar un comentario o *feedback* sobre la propuesta compuesto por el comentario en sí y una valoración o *rating* del 1 a 5. Puede enviar todos los *feedbacks* que quiera.

2.7.3. El usuario puede ver los *feedbacks* enviados por los demás usuarios así como el nº de votos a favor y en contra.

2.7.4. El usuario puede acceder a otra pantalla de estadísticas en la que ver otros datos de la propuesta así como el resultado de la votación de forma gráfica.

2.8. Dispone de un mapa en el que se muestran marcadas las zonas predefinidas en las que se pueden crear propuestas. En cada marcador se visualiza el nº de propuestas asociadas a esa zona, y el usuario puede acceder pulsando en él directamente a la pantalla principal de lista de propuestas filtrada por esa zona.

2.9. Si el usuario está autenticado, puede crear una nueva propuesta a través de un formulario.

2.10. (*) El usuario puede autenticarse a través de un botón en el menú: se hace uso del **componente AAC (*Authentication and Authorization Control System BB*) de WeLive** para la autenticación de usuarios.

2.11. (*) Dispone de una pantalla *About* en la que se muestra información de la aplicación y desde la que el usuario puede ejecutar un cuestionario sobre la app hasta cuatro veces (gracias al uso de un servicio web externo desarrollado en Tecnalia).

2.12. Dispone de un menú desde el que el usuario puede acceder a las distintas pantallas: inicio de sesión, lista de propuestas, formulario de nueva propuesta, mapa, *About* y cambio de idioma.

2.13. (*) Se hace uso del **componente *Logging BB* de WeLive** para registrar ciertos eventos en relación a la actividad de la aplicación, por ejemplo, "app iniciada" o "POI creado por el usuario x". Para ello la

aplicación debe autenticarse también ante la plataforma WeLive (distinto a la autenticación del usuario, pero a través del mismo componente AAC BB).

En esta segunda app, Bilbozkatu, la gestión de propuestas y *feedbacks* se hará de manera externa a través de dos *Building Blocks* que, a diferencia del de *Logging BB* y *AAC BB* en ambas apps (y el ODS en BilbOn), no están desarrolladas y se deben implementar dentro del proyecto fin de grado.

3. Users Feedbacks BB

    3.1. Consiste en un *Building Block* **genérico** en forma de servicio web (REST API) para la gestión de *feedbacks* de **cualquier aplicación** sobre algún *item*. Por tanto, la identificación de un *feedback* consiste en la aplicación a la que pertenece, y el objeto al que hace referencia (en este caso, "bilbozkatu" y "una propuesta".

    3.2. Un *feedback* consiste en un comentario y en un *rating* del 1 al 5.

    3.3. El BB debe permitir:

        3.3.1. Crear un *feedback*.

        3.3.2. Obtener los *feedbacks*.

        3.3.3. Obtener la media de *rating* de los *feedbacks* para un objeto en concreto.

        3.3.4. Obtener el WADL general del servicio para obtener en formato XML las funcionalidades que ofrece y la forma de uso.

    3.4. Se debe desplegar y ofrecer a través de la plataforma Cloud Foundry.

4. Bilbozkatu BB

    4.1. Consiste en un *Building Block* **específico de aplicación** en forma de servicio web (REST API) para la gestión de propuestas de Bilbozkatu.

    4.2. Una propuesta consiste en un título, zona, categoría y descripción.

    4.3. El BB debe permitir:

        4.3.1. Crear una propuesta.

        4.3.2. Obtener datos generales de propuestas (todas y en base a algunos criterios como zona, categoría y/o texto).

        4.3.3. Obtener detalles de una propuesta.

        4.3.4. Votar una propuesta por parte de un usuario.

        4.3.5. Registrar un usuario (a pesar de que está registrado en el *AAC BB* de WeLive es necesario guardar el identificador del usuario de WeLive con objeto de controlar que cada uno haya votado una vez una propuesta, ya que este es un servicio independiente).

4.3.6. Obtener el WADL general del servicio para obtener en formato XML las funcionalidades que ofrece y la forma de uso.

4.4. Se debe desplegar y ofrecer a través de la plataforma Cloud Foundry.

En resumen, se deben desarrollar dos aplicaciones que hagan uso de componentes de WeLive y en Bilbozkatu de otros dos componentes que también hay que realizar, uno genérico que pueda ser usado por futuras aplicaciones en relación a la gestión de *feedbacks* y el otro específico de la aplicación para persistir los datos relacionados con las propuestas de Bilbozkatu.

En el alcance también se incluye la **formación** necesaria en las tecnologías utilizadas para el desarrollo de los cuatro artefactos software, algo destacable y de valor añadido por la cantidad de conceptos y técnicas aprendidas de manera autónoma tanto para el desarrollo de las apps como para el de los servicios web y su despliegue debido a no haberlo trabajado a lo largo de la carrera. Por ejemplo, Ionic Framework en general para las apps (que indirectamente ha provocado la necesidad de aprender y utilizar Apache Cordova para el desarrollo de apps híbridas y el *framework* AngularJS para la app en sí), así como Spring y Spring MVC para los servicios web y el uso de Cloud Foundry.

Comentario sobre la alteración del alcance

Los cambios mencionados anteriormente que han tenido un impacto en la planificación y dedicación del proyecto fin de grado, teniendo que retrasar su entrega, se han debido principalmente a cuatro factores:

1. **Cambios en las especificaciones de los componentes de WeLive.** Han sido varias las modificaciones que ha habido en las APIs de los diferentes *Building Blocks* usados de la plataforma WeLive, como el *ODS* (*Open Data Stack)* o el *Logging BB* a lo largo del desarrollo. Además, en ocasiones no se ha tenido constancia de dichos cambios lo que ha provocado una pérdida de tiempo hasta poder contactar con el desarrollador y aclarar dudas u obtener la documentación actualizada.
2. **Nuevas pantallas o funcionalidades en los servicios públicos (apps).** Inicialmente no se habían definido algunas funcionalidades que todas las apps deben incluir, como la pantalla inicial de *splash*, la de Términos y Condiciones de uso, el registro de los KPIs (*Key Performance Indicators*) definidos para cada app en el *Logging BB* o el cuestionario de la app.
3. **Añadir soporte para otra base de datos en los BB's desarrollados.** Los BBs *usersFeedbackBB* y *bilbozkatuBB* trabajaban contra una base de datos MySQL inicialmente. Al desplegarlos en la plataforma Cloud Foundry el único servicio de BD que se podía instanciar era PostgreSQL, por lo que ha habido que añadir soporte para PostgreSQL a dichos BBs, cosa que ha provocado tener que solucionar nuevos problemas.

4. **Testeos internos en Eurohelp y externos en Deusto de las apps.** En la última fase del desarrollo, personal de Eurohelp y de Deusto han realizado testeos de las apps. Esto ha hecho que, además de detectar unos pequeños *bugs*, se hayan definido algunas mejoras que realizar en las apps.

Conviene mencionar el **alcance técnico** del proyecto que consiste en el uso del Framework de Ionic para el desarrollo de las aplicaciones híbridas y en el de los componentes de WeLive a través de las APIs correspondientes. Por otro lado, para los servicios web o *Building Blocks* se utiliza el entorno J2EE con Spring y, en concreto, la herramienta Spring Boot para crear las aplicaciones Spring MVC con soporte para una base de datos MySQL y PostgreSQL. Algunas de estas tecnologías se detallarán en el apartado 5.

3.2. Exclusiones del proyecto

Inicialmente, junto con Aritz Rabadán de Eurohelp, se excluyeron del proyecto los siguientes puntos:

- Aunque el desarrollo de las apps es híbrido, de momento no se realiza el empaquetado para iOS. Si bien dicho empaquetado es directo a partir del código desarrollado gracias a Ionic (y Apache Cordova internamente), es posible que el uso de algún *plugin* provoque que haya que retocarlo para la compilación de las apps en iOS.
- En la app Bilbozkatu no se permite subir una imagen al crear una propuesta. La pantalla de detalles de propuesta mostrará una imagen predefinida asociada a la categoría a la que pertenece la misma.
- No se aplica ningún sistema de seguridad o autenticación de WeLive para el uso de los *Building Blocks* desarrollados: *usersFeedbackBB* y *bilbozkatuBB*. Esto se hubiera hecho con el componente AAC BB de WeLive.
- Tanto las apps como los *Building Blocks* desarrollados no se publican en el *marketplace* de WeLive (pero sí en Google Play las apps y en Cloud Foundry los BBs). Las tareas a realizar para la integración definitiva en la plataforma WeLive quedan excluídas del proyecto.

Capítulo 4

4. METODOLOGÍA

En este capítulo se describe la metodología de trabajo seguida y aplicada en el proyecto fin de grado. Dicha metodología es la utilizada en Eurohelp en este caso, y es por ello que se ha heredado para seguir un proceso ágil e iterativo con la ayuda de la directora del proyecto WeLive en Eurohelp, Leire Bardají.

La metodología ha servido también para conocer procesos de trabajo estandarizados en un entorno laboral y trabajar de manera colaborativa con profesionales cualificados en relación al seguimiento y control del proyecto.

En este proyecto fin de grado se ha utilizado una de las metodologías ágiles más conocidas, **SCRUM**, que mediante la aplicación de un conjunto de buenas prácticas de trabajo en equipo se potencia la obtención del mejor resultado posible de un proyecto.

SCRUM es un modelo de desarrollo ágil que permite adoptar una estrategia de desarrollo incremental frente a una planificación completa del proyecto. Esto es especialmente relevante en los proyectos de software, en los que al inicio del proyecto se dispone de poca información o de información insuficiente como para realizar una planificación completa del proyecto de inicio a fin (como ha sucedido), tal y como se hace en los proyectos tradicionales. Además, la propia naturaleza cambiante del software hace complicado poder hacer una planificación al uso, ya que a lo largo de la vida del proyecto la variación de los requisitos, o la aparición de nuevas tecnologías afectan al proyecto. Solo haciendo un seguimiento continuo se pueden conseguir proyectos de calidad, que den lugar a productos o servicios que satisfagan a los usuarios finales.

SCRUM se caracteriza por el desarrollo incremental, y por aplicar un ciclo de vida incremental e iterativo donde existe solapamiento entre las diferentes fases de desarrollo (frente a los tradicionales ciclos de vida secuenciales o en cascada). En la figura 5 se muestra el proceso iterativo que se ha ido siguiendo.

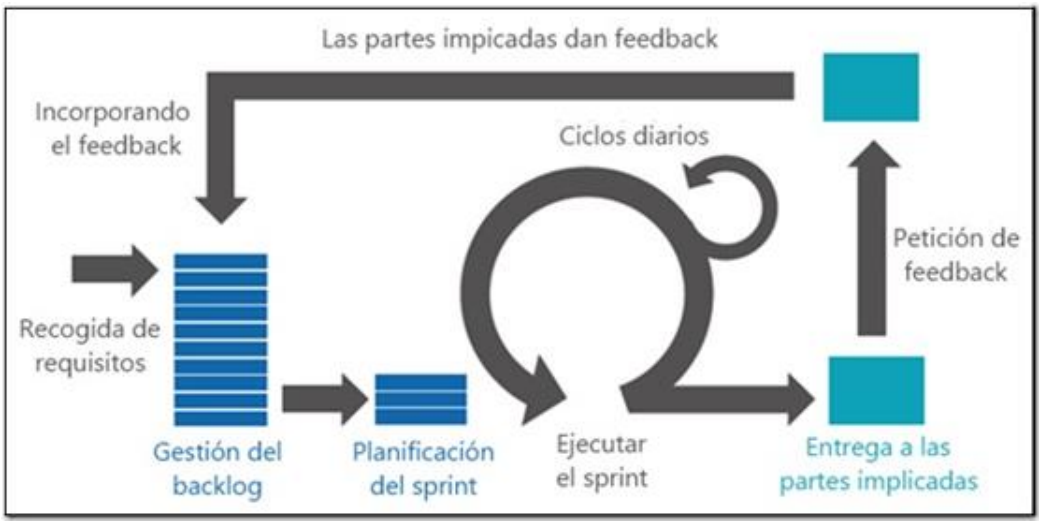


Figura 5: Flujo de gestión en SCRUM [6]

SCRUM se enmarca dentro de la "Gestión ágil de proyectos", que no siempre es aconsejable aplicar a todos los proyectos y en aquellos que se aplica se dan algunas o todas las circunstancias siguientes:

- Prioridad, satisfacer al cliente.
- Se aceptan requisitos cambiantes (como ha sucedido en este proyecto).
- Entregas frecuentes.

[6] Fuente: http://www.siderare.com/servicios/

SCRUM constituye un conjunto de prácticas enfocadas a aumentar la productividad. Es preciso, en cada organización, adaptar las prácticas SCRUM a las necesidades del contrato (enfocado a conseguir pequeños incrementos de software y a adaptarse al cambio). En la siguiente Figura 6 se puede ver la relación entre reuniones de un sprint, seguimiento y entrega.

Figura 6: Relación entre reuniones, seguimiento y entrega en SCRUM [7]

La estructura organizativa del proyecto la definirá el líder del proyecto en la fase de inicio del mismo, donde se celebra una reunión *kick-off* o de puesta en marcha en la que se designan los siguientes roles para distribuir las responsabilidades:

- ***Product Owner*** (o dueño del producto): representa al cliente (plataforma WeLive en general, y Ayto. de Bilbao en concreto), a las personas que han solicitado el producto resultante del desarrollo/mantenimiento software solicitado. Es quien marca los requisitos y gestiona la prioridad de estos.

  En este caso el Ayto. de Bilbao no ha especificado los requisitos al detalle, por lo que algunos de ellos han sido definidos en conjunto entre Aritz Rabadán de Eurohelp y el autor del presente proyecto fin de grado.

- ***Scrum Master*** (o facilitador): esta figura es la que sustituye al gestor de proyectos tradicional. Su trabajo es gestionar adecuadamente el proyecto, evitando interrupciones, controlando la variación del alcance, gestionando los requisitos, controlando los riesgos, etc. En definitiva, su labor es eliminar los obstáculos que impiden que el equipo alcance el objetivo del sprint. Se designa este rol a personas con formación y experiencia en la gestión de proyectos, con capacidad de liderazgo y de decisión. Esta persona habitualmente está dedicada al 100% al proyecto, para conseguir un seguimiento diario, evitando desviaciones que puedan afectar a la correcta consecución de los objetivos del proyecto. Este rol lo ha asumido Leire Bardají de Eurohelp.

- ***Scrum Team*** (o equipo de desarrollo): Es el grupo de personas que tiene la responsabilidad del desarrollo del producto. Por lo general debe ser

[7] Fuente: Documentación de SCRUM de la empresa Eurohelp

multidisciplinario, un conjunto de personas, dependiendo del volumen del proyecto, que puedan abarcar todas las tareas que conlleva el proyecto: análisis, diseño, desarrollo, pruebas, etc.

En el proyecto fin de grado el equipo de desarrollo ha sido compuesto exclusivamente por el autor del mismo, si bien ha recibido la colaboración puntual de Alex Novoa de Eurohelp en Bilbao en tareas de diseño (se detallará en el apartado 6.3 en relación a la app Bilbozkatu).

Existen también los roles de s***takeholders*** (clientes, proveedores, vendedores), cualquier implicado cuyos intereses puedan verse afectados como consecuencia del desarrollo del proyecto (en este caso no procede). Por último, también está la figura de los ***administradores*** (*managers*), cualquier persona que establece el ambiente para el desarrollo del producto (incluye personal de las áreas transversales de la empresa como administrativos, responsables de compras, responsables de sistemas, etc.).

Reuniones SCRUM

Uno de los principales pilares de la metodología **SCRUM** son las reuniones, ya que permiten que el equipo pueda comunicarse diariamente entre ellos y con los responsables del proyecto, manifestar el trabajo realizado y las complicaciones experimentadas, buscar soluciones en conjunto, etc.

Toda la gestión del proyecto se basa en ***Sprints*** (iteraciones). Un sprint es un periodo de duración constante (definido al inicio del proyecto) que establece el tiempo disponible para generar un entregable. Al final de cada sprint el equipo (en este caso el desarrollador, es decir, el autor del proyecto de fin de grado) presenta los avances y resultados logrados, que constituyan un entregable para el cliente, aunque en este proyecto no se ha entregado ningún artefacto intermedio a ningún cliente. Normalmente, en los proyectos TIC se establece una duración de **15 días naturales** para los *sprints* (adaptables dependiendo del proyecto). Dicha periodicidad se marca en función de las características de cada proyecto (a diferencia de en el proyecto de fin de grado, en un proyecto desarrollado para la asignatura de Ingeniería del Software dichos *sprints* han durado unas 3-4 semanas por ejemplo).

Existen diferentes tipos de reuniones en la metodología SCRUM, aunque en este caso no se han aplicado todas. Por ejemplo:

- ***Daily Scrum* o *Stand-up meeting***: Reunión diaria que se celebra cada día del *sprint*. Son reuniones cortas (aprox. 15 minutos). Cada miembro debe contestar a tres preguntas: "¿Qué ha hecho desde ayer?", "¿Qué harás hasta la reunión de mañana?", "¿Has tenido algún problema que te haya impedido alcanzar tu objetivo?". El objetivo es que todos conozcan el trabajo realizado, qué se va a realizar, y tratar de solventar los problemas o impedimentos surgidos.

- **Reunión de Planificación del Sprint (*Sprint Planning Meeting*)**: Al inicio de cada ciclo de *Sprint* se lleva a cabo una reunión de planificación del *Sprint* para identificar el trabajo a realizar y preparar el *Sprint backlog* que detalla las tareas y el tiempo asignado para su resolución.
- **Reunión de Revisión del Sprint (*Sprint Review Meeting*)**: Reunión que se realiza al final del ciclo *Sprint* para revisar el trabajo completado y no completado, preparar una demostración al cliente del trabajo realizado.
- **Retrospectiva del Sprint (*Sprint Retrospective*):** Una vez concluido cada *Sprint*, se realiza una reflexión sobre el *Sprint* recién superado, en la cual los miembros del equipo dejan sus opiniones sobre lo ocurrido en dicho Sprint. Todo ello con único y principal objetivo de obtener una mejora continua del proceso. Esta reunión tiene una duración no mayor a las tres horas.

En el proyecto fin de grado, sin embargo, al tratarse de un equipo reducido de personas (la directora, el desarrollador, y el intermediario con otros responsables de WeLive en Europa con el que también se comentan aspectos técnicos del desarrollo) la periodicidad de reuniones ha sido de 1 a 2 semanas. En dichas reuniones (que se pueden considerar *Spring Planning Meetings* o Reuniones de Planificación del Sprint, normalmente de 30-45 minutos) se han comentado los avances en el proyecto respecto a la anterior reunión y los problemas encontrados, y después se han planificado nuevos objetivos de cara a la siguiente reunión. Dichos objetivos o lista de tareas normalmente se recogen en el *Sprint Backlog*, aunque en este caso se han registrado en las respectivas actas de reuniones realizadas por el autor del presente proyecto.

También ha habido una reunión extraordinaria para comentar el *feedback* de los que se han encargado de probar las aplicaciones en la última parte de la fase de desarrollo de las mismas, tanto por parte interna en Eurohelp como por parte externa en Deusto.

Herramienta de gestión del proyecto: **JIRA**

Finalmente, cabe destacar la herramienta utilizada en la empresa para la gestión del proyecto de fin de grado: JIRA. Es una aplicación basada en web que sirve, entre otras muchas cosas, para crear y asignar a alguien, llevar un registro de trazas de trabajo (indicando la fecha, hora, duración y descripción de la traza) de cada una de ellas, realizar un seguimiento de incidentes, marcar las tareas como terminadas o no y poder visualizarlo de diferentes modos como en un panel o *dashboard* con las áreas *to do*, *in progress* y *done...*

La herramienta también ofrece paneles de control adaptables o estadísticas, como por ejemplo el *Time Sheet Report* en el que poder visualizar todas las trazas registradas en el proyecto entre dos fechas o la dedicación total diaria.

Capítulo 5

5. HERRAMIENTAS Y TECNOLOGÍAS

En este capítulo se describen y analizan algunos aspectos destacables del proyecto en relación a los servicios y tecnologías utilizadas y en las que ha sido necesaria una formación previa de manera autónoma. Se realiza un repaso al Framework de Ionic utilizado en las apps y se describen los componentes de WeLive utilizados.

Por otro lado se mencionan a grandes rasgos otras tecnologías y herramientas utilizadas en el desarrollo de los artefactos software del proyecto fin de grado, pero sin entrar demasiado en detalle en ellas.

5.1. Ionic Framework

Figura 7: Logo de Ionic Framework [8]

Asumiendo un desarrollo híbrido de las apps, se ha utilizado el kit de desarrollo **Ionic** para la realización de aplicaciones que puedan ser empaquetadas en Android e iOS, entre otros sistemas operativos. Ionic es un potente SDK o *Software Development Kit* de código abierto y gratuito, cuya versión 1.0 salió en mayo de 2015, que facilita la creación de aplicaciones móviles con un aspecto y rendimiento muy cercanas a las aplicaciones nativas mediante el uso de tecnologías web como HTML5, CSS3 y JavaScript. Está construido con Sass y optimizado para el *framework* Angular JS que permite desarrollar aplicaciones ricas y robustas. Además, trabaja bajo Apache Cordova o Phonegap para compilar de manera nativa, siendo el modo de hacerlo muy similar al utilizado en apps hechas con Cordova, bajo el lema "desarrollar una vez, compilar muchas veces". [8]

Su objetivo principal es atajar las cuestiones relacionadas con la apariencia y la interacción de interfaz de usuario de la aplicación. Se centra en minimizar la diferencia con aplicaciones nativas implementando cuestiones como:

- Gestión del doble *tap* por defecto. Cuando se utiliza un navegador web desde un dispositivo móvil, al realizar una pulsación (*tap*) sobre un elemento de pantalla, el navegador espera hasta 200 milisegundos por si se va a realizar una pulsación doble (doble *tap*). En aplicaciones Ionic este tiempo se elimina acortando los tiempos de respuesta ante las interacciones del usuario. Al final, no existen dobles pulsaciones en las aplicaciones móviles, todo es accesible mediante una única pulsación.
- Recicla y reúsa elementos del DOM para permitir el *scroll* suave de grandes listas de elementos: la manipulación del árbol DOM de los documentos HTML es una de las principales limitaciones de rendimiento en dispositivos móviles.
- Provee una opción para utilizar por defecto un *scroll* nativo, acercando la experiencia de usuario al límite a una aplicación nativa.
- Provee animaciones aceleradas por hardware para optimizar el rendimiento en los dispositivos móviles.
- Provee por defecto elementos de pantalla (componentes UI) listos para utilizarse y fácilmente personalizables en diseño (tamaño, color, etc.) como menús deslizantes, botoneras y todo tipo de controles *input*,

[8] Fuente: https://en.wikipedia.org/wiki/Ionic_(mobile_app_framework)

componentes de formulario, barras de navegación, *pop-ups*, pestañas, listas, etc. También ofrece otras utilidades como colores por defecto o una serie de iconos fácilmente utilizables e integrables en la interfaz.

- Hace uso de *plugins* que permiten el acceso a funcionalidades de los dispositivos móviles como cámara, acelerómetro, agenda de contactos, GPS, información del dispositivo, etc.

Si bien Ionic ofrece un *framework* de desarrollo en cuanto a componentes y CSS optimizados para móviles, también enmarca una librería del UI (*User Interface*) JavaScript, necesaria para producir los efectos visuales personalizables en dichos componentes o usar *ngCordova*. Y es que una de las grandes ventajas de Ionic es que se basa en *frameworks* de desarrollo ampliamente utilizados como son **AngularJS** y **Cordova,** tal y como se ha mencionado. Ionic es la capa que está por encima de AngularJS, y se puede usar AngularJS para acceder a *plugins* nativos de Cordova (*inAppBrowser, geolocation,…*) a través de ***ngCordova***, el *framework* que deja disponible la API de Cordova a través de servicios AngularJS. La arquitectura de una aplicación Ionic se puede ver en la Figura 8.

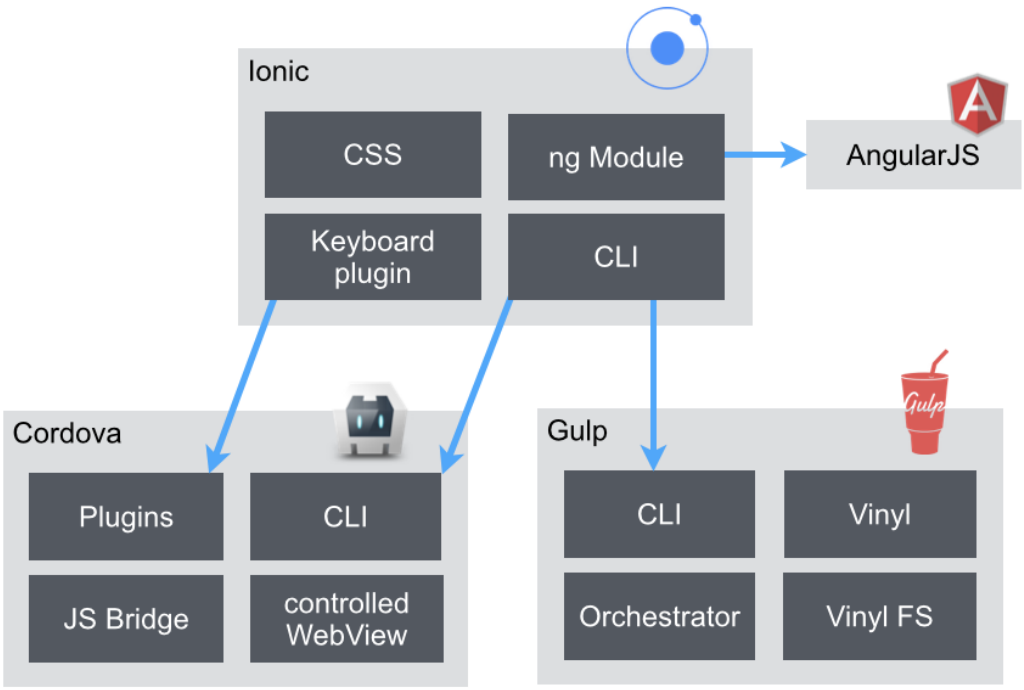


Figura 8: Arquitectura de una aplicación con Ionic [8]

En la anterior Figura 8 se puede ver cómo Ionic utiliza AngularJS para el desarrollo de las aplicaciones (se podría utilizar otro *framework*, pero en ese caso no se podría aprovechar toda la potencia de Ionic), y cómo usa Cordova para empaquetar las apps. Para comprender mejor el uso de *plugins* de Cordova con Ionic a continuación se muestra la arquitectura en la Figura 9.

[9] Fuente: http://www.slideshare.net/lucio_grenzi/use-ionic-framework-to-develop-mobile-application

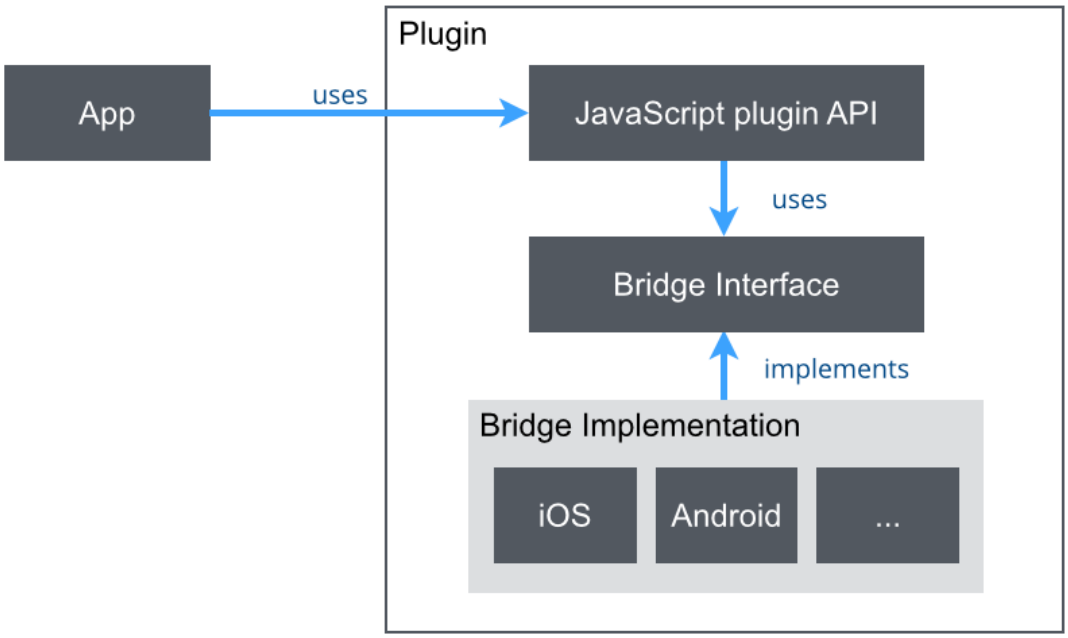


Figura 9: Arquitectura de los plugins de Apache Cordova [10]

En la Figura 9 se aprecia la arquitectura que muestra la **parte híbrida del desarrollo**. Una misma API provee todo lo necesario para que una determinada característica del dispositivo móvil, como el navegador interno de una aplicación (*inAppBrowser*), el *GPS* o la brújula, por ejemplo, pueda ser utilizada y accedida de manera idéntica para varias plataformas y con tecnologías web. Sin embargo, hay que tener en cuenta que en ocasiones algunos *plugins* no están completamente soportados para todas las plataformas, y es necesario revisar la especificación de los mismos antes de integrarlos en una aplicación. A veces los *plugins* requieren de un uso ligeramente distinto dependiendo de la plaforma, o directamente no está soportado por una de ellas.

Framework Apache Cordova y su origen

Conviene comentar el *framework* de desarrollo Cordova del que Ionic hace uso, el cual usa tecnologías web estándares y desarrollo en JavaScript para su empaquetado en distintas plataformas. En este caso, Ionic está más orientado al *front-end* de la aplicación, los componentes CSS para la interfaz, aunque dispone de más utilidades dentro de su librería JavaScript como el control del historial de las pantallas "visitadas" en la aplicación en relación a la navegación.

En su origen este *framework* se llamaba PhoneGap, con la idea de crear apps orientadas a móviles con HTML5 pero con una capa JavaScript que permita acceder a las funciones nativas del dispositivo de cualquier plataforma. Debido a su éxito, la empresa Nitobi que lo creó buscó colaboradores y así en 2011 donó el código del producto a la fundación Apache, convirtiéndolo en un proyecto

[10] : Fuente: http://hsc.com/Blog/Enterprise-Mobile-Application-Development-Trends-and-Best-Practices

Open Source. Poco después la empresa Adobe compró Nitobi, haciéndose por tanto con PhoneGap. Sin embargo, Adobe mantuvo los acuerdos en relación a la donación del código a Apache.

Actualmente PhoneGap y Apache Cordova son prácticamente idénticas, pero PhoneGap es una marca registrada de Adobe y si quisiera podría explotar el producto con herramientas añadidas propias. Es por ello que Apache decidió cambiar el nombre del producto a Cordova, para diferenciarlo, en febrero de 2012.

Para ilustrar a alto nivel la arquitectura de Apache Cordova, en la siguiente Figura 10 se aprecia cómo está enfocado a tecnologías web. Crea una estructura de app nativa y dentro de ella abre un componente que actúa como motor de renderizado HTML, *WebView*, y tiene una capa JavaScript que ofrece una API que soporta la interacción y uso de APIs específicas de plataforma. Como se acaba de comentar, las aplicaciones Cordova se ejecutan dentro de un *WebView* controlado por Cordova.

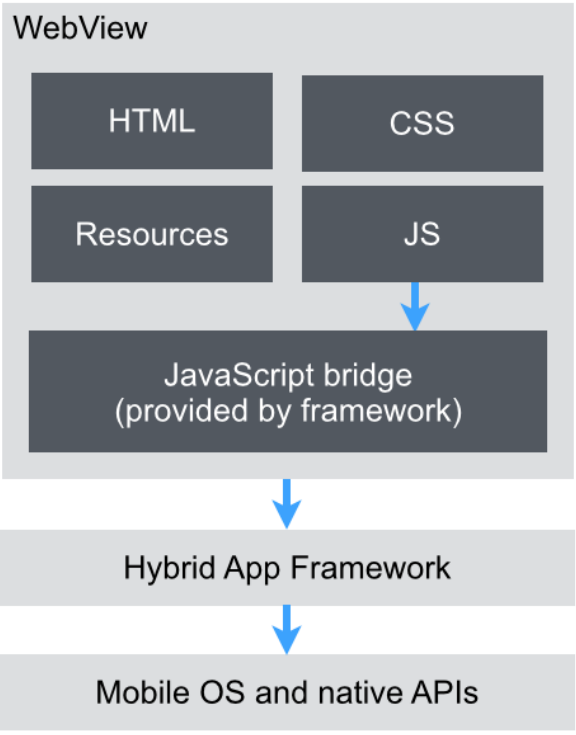


Figura 10: Arquitectura a alto nivel de Apache Cordova [11]

Los *plugins* pueden ser extendidos, modificados, o incluso es posible crear nuevos *plugins* de Cordova para atajar necesidades propias de la aplicación no contempladas en los *plugins* oficiales. En este caso no ha sido necesario, y los plugins utilizados en las aplicaciones se detallan en los apartados del capítulo 6 correspondientes a las aplicaciones móviles desarrolladas.

[11] : Fuente: https://blog.codecentric.de/en/2014/11/ionic-angularjs-framework-on-the-rise/

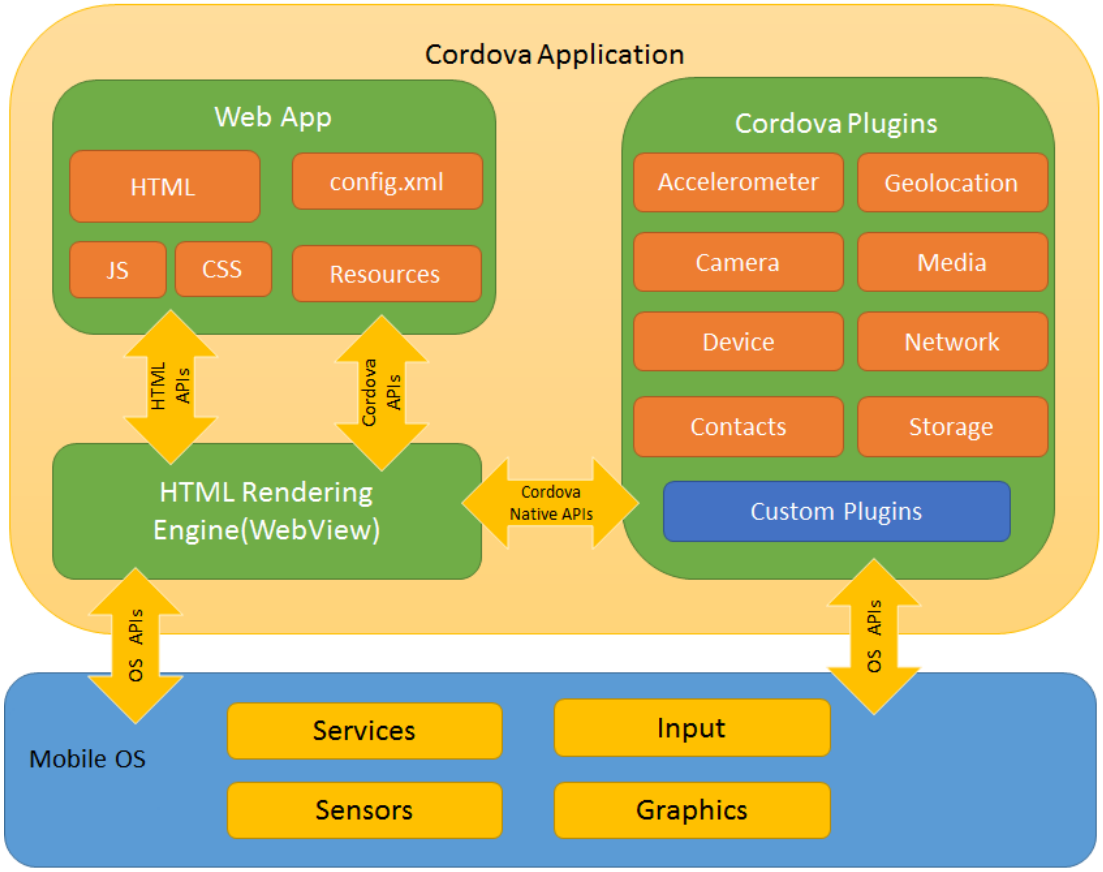


Figura 11: Arquitectura detallada de aplicaciones Cordova [12]

En esta Figura 11 se muestra más detalladamente la arquitectura de una aplicación Cordova. El *Web App* es donde el código desarrollado para la aplicación reside como si fuera el de una página web. En dicho código el fichero principal y necesario para la ejecución es el ***index.html*** (igualmente sucede con Ionic), y en él se referencian los demás ficheros HTML, CSS, JavaScript y de recursos.

Otro fichero crucial es el llamado ***config.xml***, en el cual se indica la información sobre la aplicación (id, versión, autor, etc.) y se configuran algunos parámetros que afectan al modo de funcionamiento (permisos, plugins, orientación de la app en el dispositivo, etc.).

Framework AngularJS

Respecto a AngularJS, Ionic básicamente lo extiende con una serie de elementos que facilitan, aún más, el desarrollo de las aplicaciones. AngularJS es el *framework* JavaScript de código abierto y gratuito más popular en la actualidad. Está respaldado por Google y cuenta con una vibrante comunidad de desarrolladores que pone a disposición pública multitud de recursos didácticos y códigos de ejemplo.

[12] Fuente: https://cordova.apache.org/docs/en/latest/guide/overview/

AngularJS implementa un paradigma de programación conocido como *modelo-vista-cualquier cosa* (*Model-View-Whatever*). Este modelo permite una separación de responsabilidades y un mantenimiento de la aplicación muy sencillo. La siguiente Figura 12 lo ilustra muy bien.

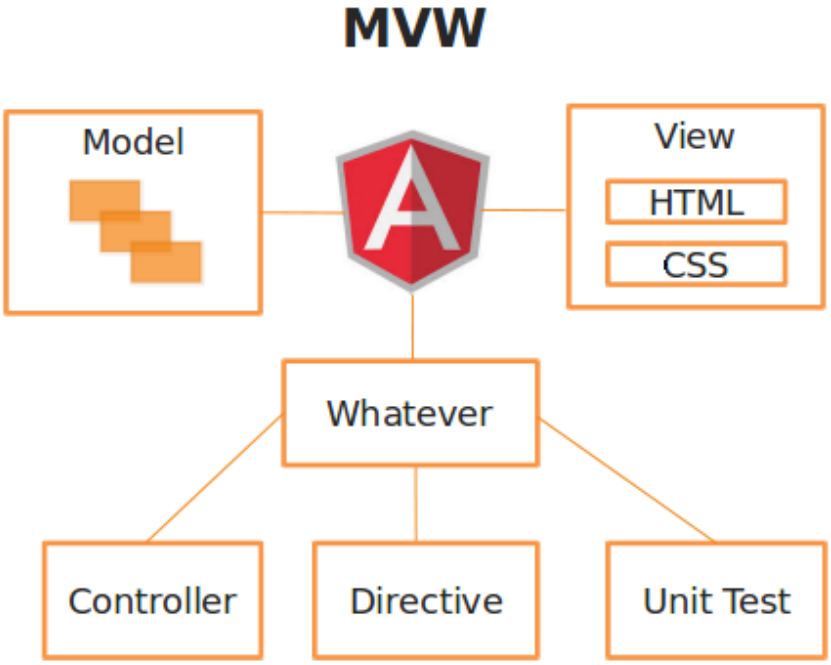


Figura 12: Paradigma de programación MVW de AngularJS [13]

En la Figura 12, entre el contexto Angular y "*Whatever*" se sitúa algo llamado "*two-way data binding*", que permite que las actualizaciones sobre el modelo se reflejan automáticamente en la vista (típicamente documentos HTML) y viceversa, lo que facilita enormemente esa interacción entre el modelo y la vista. Por ejemplo, el desarrollador sólo debe preocuparse en generar una **plantilla** HTML, en la que gracias a la anotación específica de Angular inserta, en el lugar de la vista que quiera, el dato del modelo que desee. De este modo, tanto si es el usuario el que modifica ese dato (por ejemplo, si ese dato está asociado a una caja de texto) o si se modifica desde el controlador, automáticamente dicho cambio se verá en el otro lado.

Esto se realiza gracias a la característica de Angular llamada ***$scope***, un elemento en el que almacenar funciones y variables en el contexto de un controlador asociado a una vista, y puede ser construido de manera jerárquica entre los mismos. También existen otras muchas características, como filtros, servicios, directivas (etiquetas o atributos propios que extienden el vocabulario HTML y sirven para añadir funcionalidades a los elementos, que favorece la orientación al uso de componentes reutilizables), routing entre vistas, testing,...

[13] Fuente: http://www.slideshare.net/EdurekaIN/angular-js-wp29july

pero todo esto viene bien detallado en la documentación oficial de AngularJS: https://docs.angularjs.org/guide.

El *framework* Ionic, a pesar de que se ha especificado como *modelo-vista-cualquier cosa* (*MVW*, *Model-View-Whatever*), también se suele considerar *modelo-vista-controlador* directamente (*MVC*, *Model-View-Controller*) como se representa a continuación en la Figura 13, e incluso *modelo-vista-vistaModelo* (*MVVM*, *Model-View-ViewModel*) (el objeto *$scope* se puede considerar VistaModelo) con las muchas mejoras que han ido teniendo.

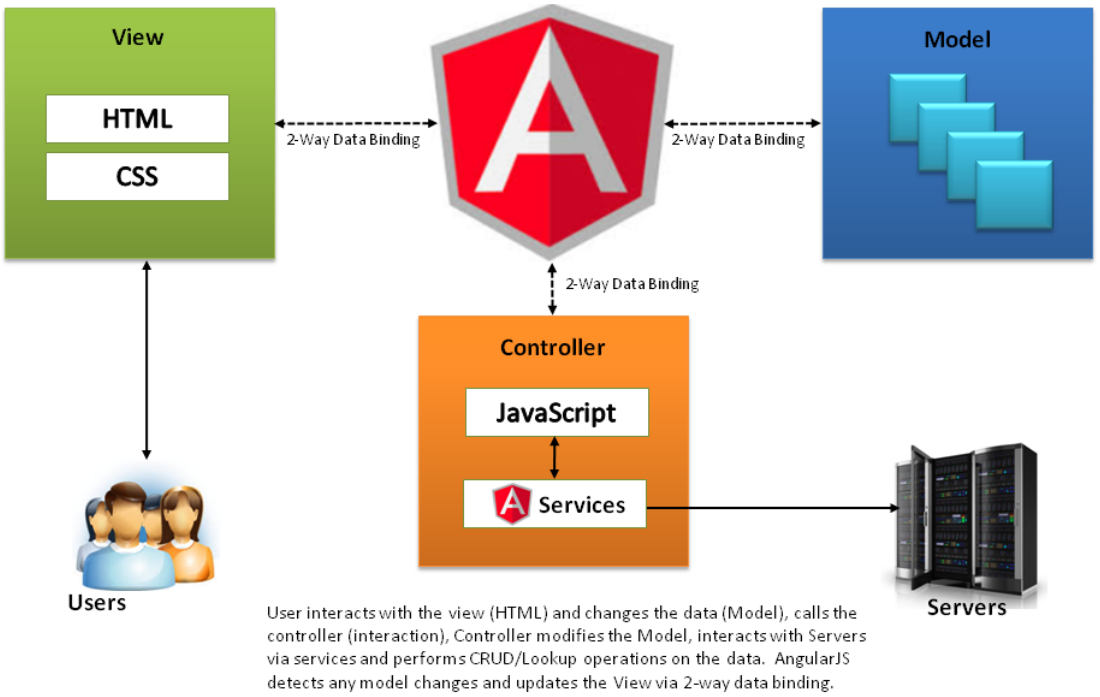


Figura 13: Paradigma de programación MVC de AngularJS [14]

En resumen, los conceptos principales de AngularJS son el *scope* (lo que une el modelo o los datos de la aplicación y su comportamiento), las directivas (extensiones de HTML para crear etiquetas y atributos con funcionalidad personalizable) y los servicios (lógica del negocio reutilizable e independiente de las vistas).

Ecosistema de productos de Ionic

Además del propio *framework* de desarrollo, Ionic cuenta con un **ecosistema de productos** que facilita el ciclo de vida del proyecto, algunos de los cuales han sido utilizados en el proyecto de fin de grado:

- ***Ionic Creator***: Permite la definición de pantallas mediante técnica *drag&drop*: se arrastran los elementos deseados sobre un lienzo y luego

[14] Fuente: http://www.slideshare.net/EdurekaIN/angular-js-wp29july

es posible exportar el conjunto de páginas definidas, así como la navegación entre ellas, como aplicación Ionic.

- ***Ionic View:*** se trata de una aplicación móvil que permite visualizar aplicaciones Ionic. El funcionamiento es el siguiente: un desarrollador, a través de una cuenta registrada en Ionic.io, sube la aplicación a su cuenta en la nube. Hecho esto, tiene la posibilidad de enviar una invitación a cualquier otro usuario de Ionic.io para que pueda, previa autenticación en Ionic View, aceptar la invitación y abrir e interactuar con la app Ionic. Esta funcionalidad es muy útil para distribuir versiones de la aplicación a testers y clientes antes de hacerla pública en el *store* correspondiente (Google Play o App Store en este proyecto de fin de grado) una versión final estable.

  Es importante tener en cuenta que Ionic View solo soporta ciertos *plugins* de Cordova, por lo que hay que tenerlo en cuenta a la hora de probar una app. En ese caso habría que probar la app con algún emulador o empaquetando en instalando la aplicación en un dispositivo móvil, ya que en el navegador, si bien se pueden testear muchas funcionalidades gracias a las tecnologías web que se usan, el acceso a características como la brújula o el acelerómetro de un dispositivo no está disponible. La lista de *plugins* soportados por Ionic View se puede consultar en la página http://docs.ionic.io/docs/view-usage.

- ***Ionic CLI:*** se trata de una serie de comandos ejecutables en un terminal que permiten la compilación, empaquetado y visualización de una aplicación Ionic en desarrollo.

- ***Ionic Labs:*** ofrece las mismas funcionalidades que Ionic CLI pero desde una aplicación de escritorio con una interfaz visual rica, amigable e interactiva.

- ***Ionic Package:*** permite la generación de empaquetados para las plataformas Android e iOS mediante un servicio en la nube, sin la necesidad de contar con el entorno adecuado para cada plataforma.

- ***Push notifications:*** implementación de notificaciones originadas desde el servidor y que son recibidas por las aplicaciones móviles Ionic. Ejemplos de notificaciones *push* son: mensajes de WhatsApp, avisos de recepción de un nuevo e-mail, nuevo mensaje en Facebook, etc.

En este proyecto fin de grado se ha hecho uso tanto de **Ionic View** para compartir la app y visualizarla en dispositivos móviles como de **Ionic CLI** para realizar ciertas acciones como instalación de *plugins* de Cordova, ejecución de un servidor local para testear la app en el navegador del ordenador con cambios en caliente, o compilación y empaquetado de la app para Android.

Tal y como se especifica en el apartado 6.3, el inicio del desarrollo de la app de Bilbozkatu fue a partir de una simple carcasa de la app realizada con la herramienta **Ionic Creator** por Alex Novoa en Bilbao, de Eurohelp. Posteriores cambios en el diseño por parte del autor del presente proyecto han sido a través de código, no con dicha herramienta.

Finalmente, cabe añadir que el uso de Ionic requiere de ciertas instalaciones como **Node.js** (con el que se instala su gestor de paquetes **NPM**), **Bower** (gestor de paquetes de JavaScript) y el propio **Ionic**. Los últimos dos se pueden instalar con *npm*, aunque la instalación de Ionic requiere de algunas instalaciones previas como **Git** (o cualquier otro sistema de control de versiones), **Apache Cordova**, **Android SDK** (con *Android SDK Tools* es suficiente, no hace falta instalar todo *Android SDK Studio*) o la última versión de **Java** (requiere el *Java Development Kit* o JDK, no sólo el *Java Runtime Environment* o JRE).

En algunas instalaciones como la de Android SDK o Java JDK hay que seguir las indicaciones especificadas en la guía de instalación para configurar correctamente las variables del entorno del sistema operativo donde se esté instalando, como JAVA_HOME, PATH o ANDROID_HOME.

Más información sobre Ionic en http://ionicframework.com/ y http://ionic.io/.

5.2. Plataforma WeLive

Tal y como se ha indicado en el apartado 3.1 sobre el alcance del proyecto, las aplicaciones móviles desarrolladas hacen uso de algunos **componentes de la plataforma WeLive**, desarrollados por diferentes empresas de manera colaborativa en Europa. Dichos componentes están disponibles en forma de servicio web, en concreto proveen una **REST API**.

A continuación se enumeran y describen los componentes utilizados en las aplicaciones objeto de este proyecto de fin de grado (hay más), y una descripción general de ellos (no se incluyen detalles de los mismos en este documento, tan solo la información general que sirva para contextualizar su uso e implementación en las aplicaciones).

5.2.1. **AAC BB**

El *Authentication and Authorizacion Control System BB* (en Adelante, AAC BB o AAC) es un sistema de autorización y autenticación que se basa en el protocolo OAuth2.0 para la autorización de acceso a recursos y en los proveedores de identidad externos para la autenticación de usuarios. Específicamente, AAC soporta la integración con los proveedores de identidad disponibles a través de protocolos estándares de autorización, como Shibboleth u Open ID. La

configuración específica y el uso del AAC dependen del escenario en donde se vaya a explotar.

El protocolo OAuth2.0 define un **modelo de permisos** y los flujos de datos correspondientes de este modo:

- Cada **operación o recurso protegido está asociado a un permiso** (***scope*** en la terminología de OAuth2.0), como por ejemplo los *scope's 'profile.read' y 'profile.write'* para las operaciones de leer y modificar perfiles de un usuario. El que realiza la llamada a la operación debe tener el correspondiente permiso, autorizado por el usuario y asociado al *token* de acceso.
- Los ***scope's*** o permisos **deben referirse a operaciones de un usuario** (como modificar el perfil del usuario que ha iniciado sesión), **a operaciones genéricas** (como leer los perfiles de todos los usuarios), **o a las dos**. Diferentes **flujos de datos** del protocolo de OAuth2.0 estarán implicados en el proceso de login en función del *scope* al que el cliente quiera acceder:
    - En el caso de operaciones relacionadas con un usuario (***OAuth2.0 User-Related Protocol Flow***), el acceso debe estar concedido por el usuario autenticado con el flujo implícito (***Implicit Flow***) o con el flujo de código de autorización (***Authorization Code Flow***). Este último tiene una pequeña variación para su uso en aplicaciones móviles, y el flujo se denomina *Mobile Client Access* (aunque, como se ha dicho, es prácticamente idéntico al *Authorization Code Flow*).
    - En cuanto a los *scope's* de operaciones genéricas ningún usuario participa en el proceso (***OAuth2.0 Non User-Related Protocol Flow***), es el flujo de credenciales el necesario (***Client Credentials Flow***).

  Es importante señalar, en cuanto al modelo de permisos del protocolo OAuth2.0, que los *tokens* de acceso emitidos por dicho protocolo expiran al cabo de un corto periodo de tiempo (*short-living*). En el caso de tener que usar recursos protegidos de manera continua o en un segundo plano por detrás del usuario, pero el *token* ha expirado, es posible utilizar un *refresh-token* (obtenido al término del flujo de autenticación correspondiente) para refrescar el *token* expirado sin necesidad de repetir todo el proceso de nuevo.

  En las **aplicaciones** desarrolladas en el proyecto fin de grado, por un lado, se ha utilizado el ***Authorization Code Flow*** **para el proceso de inicio de sesión de un usuario** (con la variación comentada para el *Mobile Client Access*) con el que obtener el ***token*** **de acceso del usuario**. Dicho *token* sirve para obtener el perfil del usuario, entre otras cosas.

  Por otro lado, se ha empleado el ***Client Credentials Flow*** **para obtener el *token* de acceso de la propia aplicación** (cliente WeLive, sin implicar a

ningún usuario) para poder hacer **uso del *Logging BB*** que en el apartado 5.2.2 se comenta y que requiere de autenticación para hacerlo.

En el apartado **6.3.2.4 en relación a diagramas de secuencia de la app** Bilbozkatu se analizará mejor el **proceso seguido para obtener el *token*** de acceso de un usuario y el de una aplicación, para hacer uso de distintos recursos protegidos de la plataforma WeLive.

Tal y como se ha especificado en el apartado 3.2 en referencia a las exclusiones del proyecto, se debe recordar que no se ha incluido en el alcance del mismo el proteger los *Building Blocks* desarrollados con *scope's* del AAC mediante el mecanismo de autorización de OAuth2.0. Esto tiene que ver también con la exclusión de publicar los BBs en el *marketplace* de WeLive (sobre todo el BB o servicio *UsersFeedbackBB* genérico pensado para su uso en varias apps).

AAC REST API

En lo que respecta al API del componente AAC de WeLive, se debe comentar que dicha API se divide en:

- **API de generación de *token***. Por ejemplo, para generar un *token* de acceso (a nivel de usuario o de app), tanto a través de un proceso o flujo específico, o directamente refrescando un *token* existente.
- **API de validación de un *token*.**
- **API de datos del usuario**. Por ejemplo, para leer los datos del perfil o de la cuenta del usuario que ha iniciado sesión y tiene un *token* de acceso para ello.

En el siguiente ejemplo se muestra el JSON de respuesta al pedir **leer los datos del perfil** de un usuario, enviando para ello el *token* de acceso del mismo obtenido después del flujo de inicio de sesión correspondiente:


```
{
        "name": "Mario",
        "surname": "Rossi",
        "userId": "6789"
}
```


Figura 14: Ejemplo JSON de datos del perfil de un usuario obtenidos del AAC BB

Finalmente, para poder desarrollar una aplicación que vaya a necesitar acceso a *Building Blocks* y recursos protegidos de WeLive, tanto por parte de un usuario como por el de la propia app, es necesario **registrar la aplicación en la plataforma WeLive** (en concreto, en la consola de desarrolladores de WeLive https://dev.welive.eu/aac/dev una vez iniciada la sesión en dicha consola) como una aplicación cliente del AAC.

Una vez registrada la app, se deben seleccionar los proveedores de identificación que se quieran en la app (Google, Facebook y/o WeLive) y configurar (activar) los permisos necesarios como los del *Basic Profile Service* (del *Core AAC Service* para gestionar perfiles básicos de usuarios, como leer los datos de perfil) o los del *Logging Service* (para activar los permisos que posibiliten leer y escribir logs en el *Logging BB* tal y como se menciona en el siguiente punto 5.2.2).

En la consola de desarrolladores se deben obtener el *clientId*, *clientSecret* y *clientSecretMobile*. Estos son los *tokens* que la plataforma WeLive ha generado automáticamente cuando se ha registrado la app, y son necesarios en los distintos procesos o flujos de autenticación descritos anteriormente para el AAC BB, de manera que en dichos procesos la plataforma, por ejemplo, valide los permisos asociados a la aplicación cuando se intenta acceder a un recurso protegido. Por ejemplo, si en la consola de desarrolladores están desactivados los permisos para el uso del *Logging BB*, cuando la aplicación trate de registrar un *log* o KPI con el *token* asignado, no tendrá los permisos necesarios para hacerlo aunque haya indicado dicho *token*.

5.2.2. **Logging BB**

El componente o *Building Block* de WeLive llamado *Logging BB* es un servicio que ofrece una serie de herramientas para el almacenamiento y análisis de eventos. Dichos eventos están asociados a diferentes actividades que suceden tanto en las aplicaciones o servicios públicos como en los *Building Blocks* de terceras partes en WeLive. Estas herramientas están en forma de APIs REST y, por tanto, están disponibles programáticamente para aplicaciones y componentes desarrollados con diversas tecnologías y lenguajes.

Por tanto, sirve para reportar información relevante de la ejecución y actividad de una app ("app iniciada", "usuario x ha iniciado sesión",...) o BB y para analizar la plataforma y el comportamiento del servicio que se controla. Para ello, hay una serie de KPIs (***Key Performance Indicators***) o Indicadores Clave de Comportamiento ya definidos para todos los componentes o aplicaciones de WeLive. Estos KPIs o eventos se dividen en:

- **Eventos del nivel del sistema**. Por ejemplo, información que llega de los componentes raíz o *core* de WeLive como el *Marketplace*, *ODS* (se comenta en el punto 5.2.3), etc. Son eventos internos de la plataforma WeLive.
- **Eventos de aplicaciones de terceros**. Para la información que llega de los BBs y las aplicaciones externas desarrolladas por terceras partes.

En las aplicaciones del proyecto de fin de grado desarrolladas se han tenido que registrar en dicho *Logging BB* los KPIs previamente definidos para las mismas (Bilbozkatu y BilbOn). A lo largo del proyecto ha habido un cambio importante en relación a su uso, que ha consistido en securizarlo. Es decir, para que una

aplicación internamente pueda registrar un KPI debe hacerlo indicando también el *token* de acceso de la propia aplicación (cliente), mencionado en el ***OAuth2.0 Non User-Related Protocol Flow*** del apartado 5.2.1.

Dicho *token*, a diferencia del *token* de acceso de un usuario, no expira. Se puede obtener tanto desde el AAC BB mediante la generación de *token* (con los parámetros correspondientes para indicar que no es un inicio de sesión de un usuario y no debe iniciar ese proceso, sino que se pide el *token* para la app sin implicar a ningún usuario), como desde la consola de desarrolladores de WeLive.

En el apartado 6.3.1 (análisis de la app Bilbozkatu) y 6.4.1 (análisis de la app BilbOn) se detallan los KPIs definidos para cada una. Dependiendo del KPI los datos a registrar en él variarán. Como muestra de ello la siguiente Figura 15 muestra un ejemplo de objeto JSON que es el que se almacena en el *Logging BB*.

```
{
        "msg": "Bilbozkatu app started [KPI.BIO.7]",
        "appId": "bilbozkatu",
        "type": "AppStarted",
        "custom_attr": {
                "appname": "Bilbozkatu"
        }
```

Figura 15: Ejemplo JSON de formato de KPI para el Logging BB

La propiedad "custom_attr" de la Figura 15 tendrá los campos propios del KPI en función de lo definido. En este caso sólo se guarda el "appname", pero otros KPIs almacenan también el id del usuario o el id y nombre de una propuesta (en Bilbozkatu). El resto de campos son comunes a todos los KPIs, y su valor puede ser cualquiera.

5.2.3. **ODS**

El último de los tres componentes de WeLive utilizados en el proyecto fin de grado (en este caso sólo en la app BilbOn) es el *Open Data Stack* (https://dev.welive.eu/ods/). Este componente se ha introducido en el apartado 2.2, pero conviene recordar que trata con los desafíos asociados a la gestión del conocimiento de una ciudad en la forma de múltiples orígenes de datos con distintos formatos. En concreto, el ODS aborda los siguientes aspectos del ecosistema de datos:

- Gestionar entidades relacionadas con los datos (*datasets*, recursos...).
- Acceder, integrar, consultar y verificar *datasets* heterogéneos.
- Combinar datos sociales, del *Open Government* y también los generados por el usuario.

En el proyecto de fin de grado, al tratarse BilbOn de una app para Bilbao, trabaja con los *datasets* proporcionados por el Ayto. de Bilbao de manera abierta (*Open Data*). En https://dev.welive.eu/ods/organization/bilbao-city-council se pueden ver los más de 150 *datasets* disponibles para la consulta de datos oficiales, en este proyecto, POIs o puntos de interés.

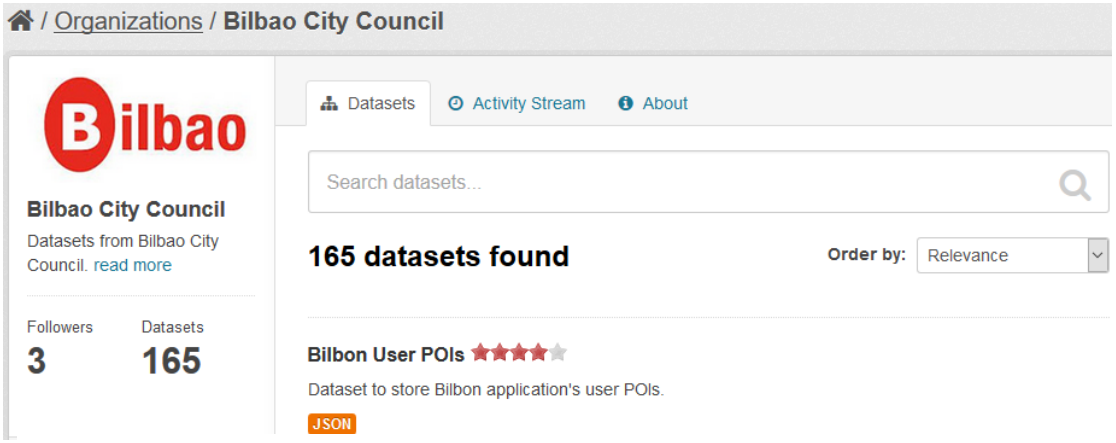


Figura 16: Interfaz del ODS para el Ayto. de Bilbao [15]

La Figura 16 muestra, a modo de pantallazo, una parte del interfaz del ODS correspondiente al ayuntamiento de Bilbao. En esa web se pueden consultar todos los *datasets* disponibles para su uso en los servicios públicos o apps. En el apartado 6.4, en el que se analiza la app BilbOn, se detallarán los *datasets* utilizados y el modo de hacerlo. En este caso serán tres *datasets* oficiales de puntos de interés de Bilbao que corresponden a tres categorías distintas, y un cuarto *dataset* (el que se aprecia en la Figura 16) creado específicamente para los puntos de interés creados por los ciudadanos a través de la app.

La consulta y modificación de dichos *datasets* se realiza a través del API REST que el ODS proporciona, y se comentará más en detalle en el apartado 6.4.3 en relación a la implementación de la app BilbOn. No obstante, cabe añadir que ese acceso se realiza indicando una sentencia SQL, ya que aunque en la Figura 16 se puede ver que los *datasets* utilizados están en formato JSON (a veces también en CSV y otros), el ODS realiza un mapeo interno a una base de datos SQLite que es a la que se hacen las consultas.

Para terminar con el punto 5.2.3, comentar que la consulta de un *dataset* a través del API del ODS no requiere autenticación, pero la modificación sí (por ejemplo, para añadir un POI de un ciudadano). Hace falta el *token* de acceso del usuario habiéndolo conseguido indicando los *scope's* o permisos necesarios. Esto se detallará, como se ha dicho, en el apartado 6.4.3. En el transcurso del proyecto en lo relativo a esto ha habido cambios en el uso de la API en cuanto a la autenticación de usuario para acceder a los recursos protegidos de WeLive.

[15] Fuente: https://dev.welive.eu/ods/organization/bilbao-city-council

5.2.4. **Swagger**

Debido a que los componentes de la plataforma WeLive han ido evolucionando a lo largo del proyecto de fin de grado, su uso también ha ido cambiando. Implementaciones que se habían hecho en las apps han tenido que ir retocándose por esa razón. Sin embargo, la documentación de ciertas APIs que el autor de esta memoria ha ido consultando no siempre ha estado actualizada, lo que ha provocado en ocasiones pérdidas de tiempo.

Para solucionarlo, aunque haya sido hacia el final de este proyecto, se ha dado a conocer el *framework Swagger* aplicado en las APIs REST de los componentes de WeLive.

NOTA
Swagger (http://swagger.io/) es un framework para APIs RESTful que ayuda en la generación de documentación y la relaciona automáticamente con la implementación, proporcionando también una representación interactiva de la API que permite un fácil testeo de las llamadas sin tener que hacerlo desde código en una app. Un cambio en la implementación se trasladará casi de manera sincronizada a dicha documentación e interfaz visual (ver el *Swagger Interface* en Figura 17), lo que resulta útil tanto a desarrolladores como a no desarrolladores.

Tabla 3: Definición de Swagger

A continuación, en la Figura 17, se muestra un ejemplo de la interfaz visual o *Swagger Interface* de la API de WeLive (https://dev.welive.eu/dev/swagger/index.html). Ya que algunas llamadas requieren de autenticación para acceder al recurso protegido mediante cabeceras o *headers* identificadas con "*authorization"* o "*authentication*" y que contienen el *token* correspondiente, se puede iniciar sesión con la cuenta que haga falta y *Swagger* automáticamente obtendrá dicho *token* y lo insertará en la llamada, por lo que no hay que preocuparse por esto.

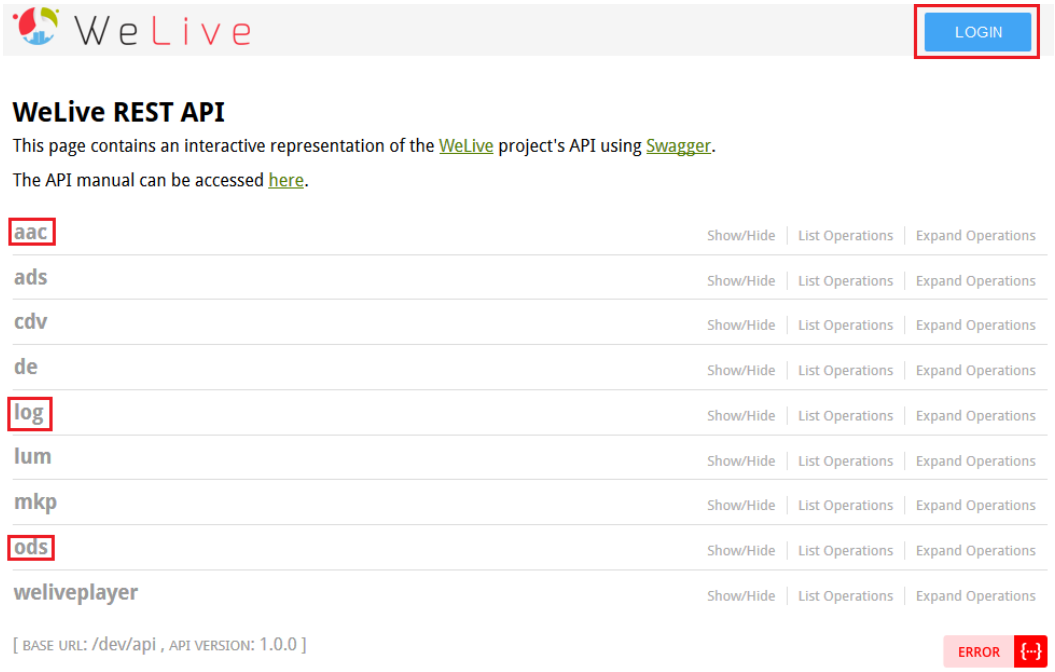


Figura 17: Representación visual de la API REST de WeLive con Swagger Interface

En la Figura 17 se puede ver, en la esquina superior derecha, el botó "Login" que permite iniciar sesión e indicar al *Swagger Interface* el *token* que va a utilizar para probar las llamadas a la API que requieran autenticación. En la parte izquierda están marcados los tres componentes utilizados en las aplicaciones desarrolladas analizados en este apartado 5.2: el AAC, *Logging BB* (*log*) y el ODS. En la siguiente Figura 18 se muestran, a modo de ejemplo, los métodos o llamadas disponibles para el componente AAC BB.

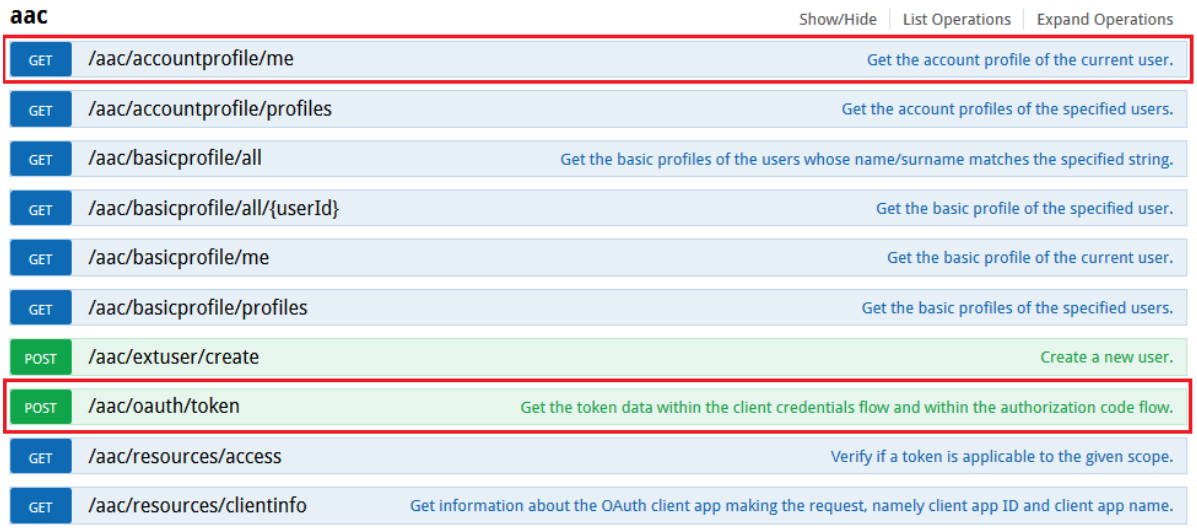


Figura 18: Opciones del AAC BB en Swagger Interface

Como se muestra en la anterior imagen, en las apps por ejemplo se utilizan los métodos "*/aac/accountprofile/me*" para obtener la información del perfil de un usuario (a través de su *token* de acceso) y sobre todo el de "*/aac/oauth/token*"

para obtener el *token* de acceso de un usuario a partir del flujo de inicio de sesión normal, refrescando un *token* ya expirado, o consiguiendo el *token* de acceso de una aplicación (cliente) de WeLive, que como se ha comentado en el apartado 5.2.1 no requiere de ningún flujo asociado a un usuario.

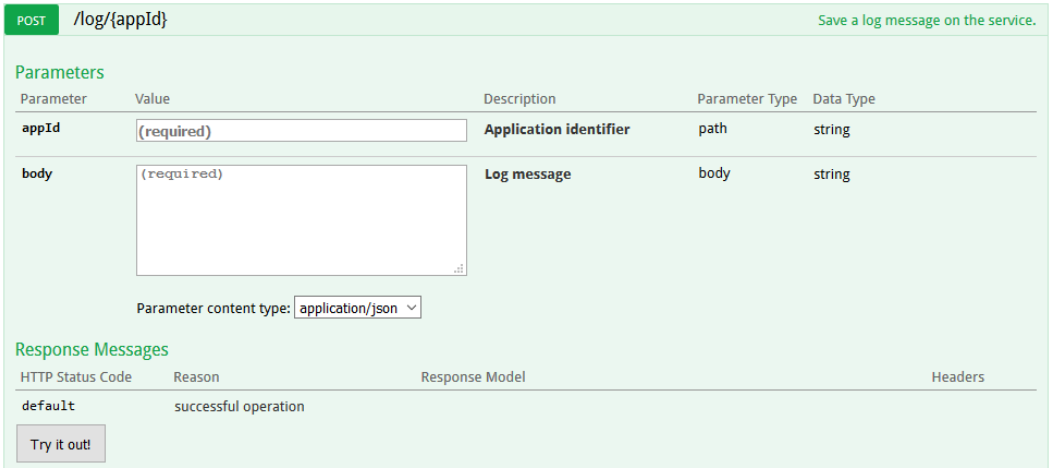


Figura 19: Método de registrar log en Logging BB con Swagger Interface

Finalmente, en la anterior Figura 19 se puede apreciar un ejemplo a la hora de realizar una llamada a la API REST de WeLive a través del *Swagger Interface*. En este caso, corresponde al componente *Logging BB* ("*/log/{appId}*") que en las apps se utiliza para registrar los diferentes KPIs definidos para ellas. En *body* se indicaría el cuerpo del evento a registrar con el formato mostrado en la Figura 15 en el apartado 5.2.2. En otro tipo de llamadas se pueden indicar los parámetros que requieran, como *client_id*, *client_secret*, *code* o *redirect_uri* en el método "*/aac/oauth/token*" del componente AAC BB.

5.3. Otras

Aunque no se vayan a detallar en profundidad como Ionic Framework o algunos componentes de la plataforma WeLive, en este proyecto fin de grado se han utilizado otras herramientas y tecnologías que a continuación se mencionan, y que en gran medida también ha sido necesaria una formación previa.

Desarrollo de *Building Blocks* (servicios web en forma de API REST)

- ***Spring Framework.*** El desarrollo de los servicios REST se ha realizado con el *framework* Spring, que es el utilizado en la empresa Eurohelp. Si bien en la asignatura de Ingeniería del Sofware II hubo una pequeña aproximación a este *framework*, en este proyecto se ha utilizado le novedosa herramienta llamada **Spring Boot**.

  Spring Boot es una tecnología dentro del mundo de Spring que permite simplificar la construcción de una aplicación Spring. Normalmente, primero se debe crear un proyecto, por ejemplo Maven, y obtener las

dependencias necesarias. Después se desarrolla la propia aplicación y, para concluir, se despliega en un servidor. Spring Boot facilita los pasos 1 y 3 para que el desarrollador se centre principalmente en el desarrollo.

Además de arrancar de manera sencilla desarrollos Spring, Spring Boot también tiene también otros objetivos como evitar la configuración vía XML (vista en la carrera) u ofrecer un CLI con algunas funcionalidades, aunque en este proyecto se ha aprovechado el IDE en su lugar. Soporta web (*Tomcat...)* y JDBC (*Java DataBase Connectivity*), entre otros.

Spring Boot se compone de varios módulos. Por ejemplo, tiene el módulo ***Spring Boot Core*** o *raíz* que sirve para el resto de módulos. En él está la clase *SpringApplication* con la que poder arrancar una aplicación Spring, o un soporte embebido para un contenedor de *servlet* ofreciendo un nuevo *ApplicationContext*, entre otras funcionalidades. Spring Boot también tiene el módulo ***Spring Boot Actuator*** (que se encarga de arrancar aplicaciones REST (como es el caso), Spring MVC, etc. fácilmente), o el módulo ***Spring Boot AutoConfigure*** para configurar la aplicación Spring programáticamente.

- **Maven**. Esta es una herramienta para la generación y gestión de dependencias de las aplicaciones Java. En el **Anexo C** adjunto al final del presente documento se describe dicha herramienta de manera un poco más detallada.
- **Cloud Foundry**. Cloud Foundry es una plataforma de código abierto disponible en la nube que se conoce como PaaS (***Platform As A Service***) desarrollada por VMWare y adquirida ahora por Pivotal. Un PaaS es un servicio de *cloud computing* que ofrece una plataforma permitiendo a los clientes desarrollar, desplegar y gestionar aplicaciones sin tener que preocuparse por la complejidad que supone la construcción y gestión de infraestructura asociada al desarrollo y despliegue de las mismas.

  En este proyecto de fin de grado se ha utilizado dicha plataforma a través de la ofrecida por la empresa FBK en Italia, que como se ha mencionado en el aparado 2.2 es una de las empresas implicadas en el proyecto WeLive, que se basa en Cloud Foundry. El uso de dicha plataforma (para subir, configurar y desplegar los *Building Blocks* desarrollados) se ha hecho a través del CLI cuya documentación es la misma que la de Cloud Foundry (http://docs.cloudfoundry.org/cf-cli/).

  Además de a través del *Command Line Interface*, también existe un *plugin* de Eclipse con el que poder acceder a una cuenta y visualizar las aplicaciones subidas. Se pueden configurar ciertos parámetros como variables del entorno y demás, y arrancar o parar las aplicaciones y ver su estado. Finalmente se ha optado por la utilización del CLI ya que funcionaba con más rapidez.

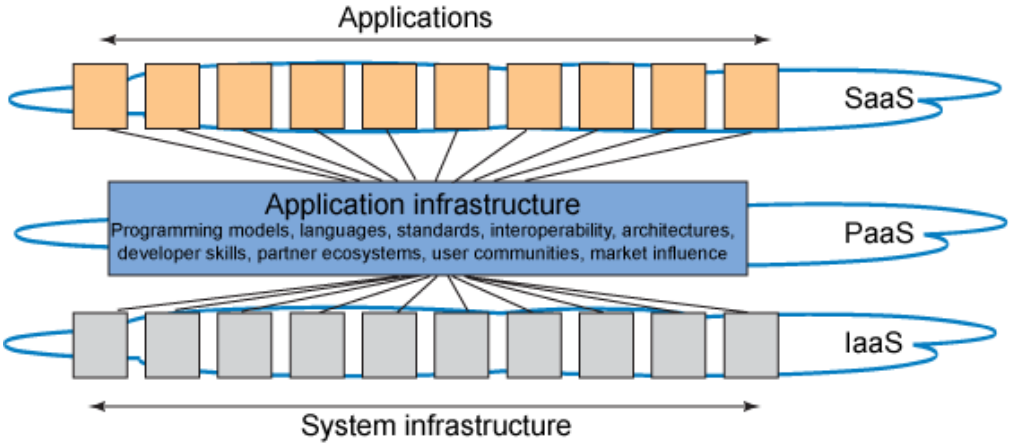


Figura 20: Infraestructura del PaaS Cloud Foundry [16]

En la Figura 20 se muestra la infraestructura de una plataforma como Cloud Foundry. La abstracción que ofrece con respecto a la infraestructura es muy útil de cara a, por ejemplo, la portabilidad entre nubes o *Clouds*. También proporciona lo necesario para *clouds* privadas, ya que algunas organizaciones no se plantean el uso de nubes públicas, y soporta múltiples lenguajes y *frameworks* en las aplicaciones. En este proyecto el lenguaje ha sido **Java**, el *framework* **Spring** y los servicios **MySQL** y **PostgreSQL** (aunque MySQL no estaba disponible en el plan contratado por WeLive para la plataforma y no ha sido usado). [17]

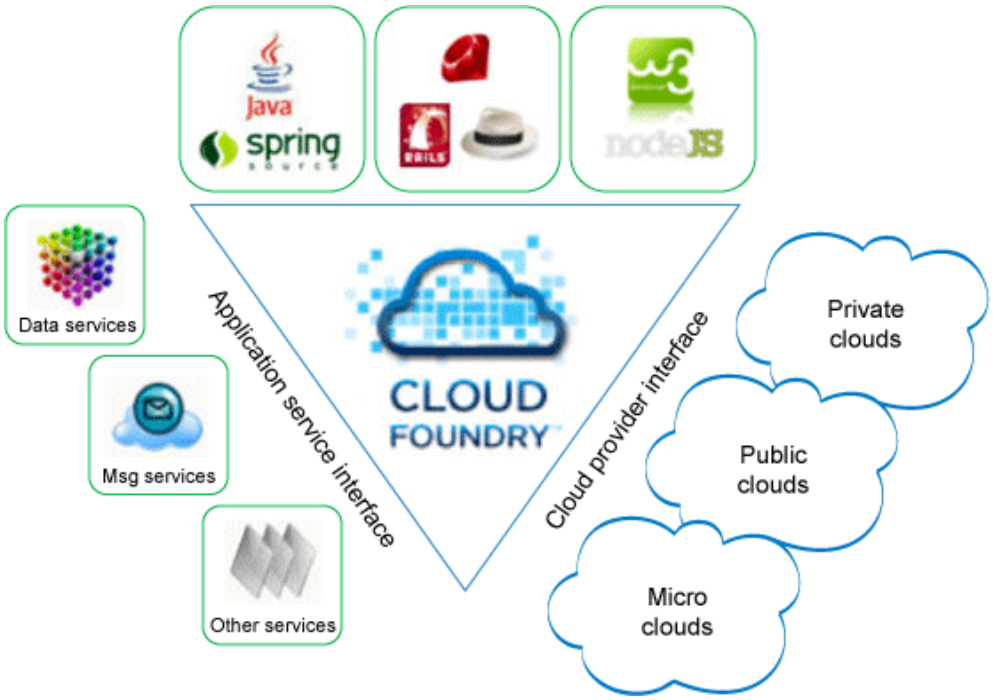


Figura 21: Soporte por capas del PaaS Cloud Foundry

[16] : Fuente: https://unpocodejava.wordpress.com/2011/11/25/que-es-cloud-foundry/
[17] Fuente: https://unpocodejava.wordpress.com/2011/11/25/que-es-cloud-foundry/

En la Figura 21 se muestra la pila de capas soportadas por *Cloud Foundry*, desde los tipos de nubes pasando por los servicios que ofrece (bases de datos MySQL, MongoDB, PostgreSQL, Redis...) y hasta los lenguajes y *frameworks* soportados (los lenguajes Java, Ruby, Node.js y Groovy, y los *frameworks* Spring, Node, Rails, Grails...).

Desarrollo de *apps*

En apartados anteriores se han analizado algunas herramientas y tecnologías utilizadas para el desarrollo de las dos aplicaciones móviles objeto de este proyecto de fin de grado. Entre ellas están, sobre todo, el *framework* Ionic (que implícitamente ha requerido formación en Ionic, AngularJS, Apache Cordova o el gestor de paquetes JavaScript Bower) y los componentes de WeLive (cuyo uso ha ido cambiando y ha habido que estudiar, además del mencionado *Swagger*).

En esta sección se ha quiere destacar también el estudio que ha requerido el uso del **API de JavaScript de Google Maps** (cuya utilización ha sido más exhaustiva en BilbOn). Ya se disponía de algo de experiencia a la hora de mostrar un mapa y, en él, marcadores. En este proyecto ha habido que extender los conocimientos para gestionar los *infoWindow* (pequeños cuadros de texto asociados a un marcador y que se muestran al pulsar en él) y, sobre todo, la herramienta **Google Places**. Esta última ofrece la funcionalidad de buscar localizaciones en base a lo que un usuario escribe en una caja de texto, mostrando una lista de sugerencias de ubicaciones en función de lo escrito.

5.4. Versiones y entornos de desarrollo

Para terminar con el apartado 5 que corresponde al análisis de herramientas y tecnologías utilizadas en el proyecto, a continuación se muestran, a grandes rasgos, las versiones de las diferentes tecnologías, librerías, entornos de desarrollo y herramientas utilizadas en el desarrollo del proyecto.

Building Blocks

- *Framework* Spring v4.2.2, con la herramienta Spring Boot v1.2.7.
- Base de datos MySQL: MySQL Community v5.7.11.0 (gestionado con MySQL Workbench v6.3.5)
- Base de datos PostgreSQL: PostgreSQL v9.5.1.1 (gestionado con el cliente pgAdmin III v1.22.1)
- Eclipse Java EE for Web Developers version Luna Service Release 2 v4.2.2 (entorno de desarrollo integrado o IDE (*Integrated Development Environment*) para el desarrollo de los proyectos Spring con Java)

El resto de versiones de las dependencias utilizadas en los *Building Blocks* (JUnit, Spring Cloud, conectores de las bases de datos...) se pueden ver en el POM

(*Project Object Model*) o *pom.xml* de los mismos, ya que utilizan Maven para la gestión de dependencias.

Aplicaciones móviles híbridas

- Ionic CLI v1.7.14 (*Command Line Interface* de Ionic)
- Apache Cordova CLI v6.1.1 (CLI de Cordova para gestión de plugins, etc.)
- Ionic App Lib v0.7.0 (Librería de Ionic) (para HTML5, CSS, JS)
    - AngularJS v1.4.3
- Node.js v4.4.6 (requerido por Ionic)
- NPM v3.10.2 (gestor de paquetes)
- Bower v1.7.9 (gestor de paquetes para librerías JavaScript)
- Java JDK 8
- Android SDK Tools v r24.4.1 para Windows (para empaquetar las apps para Android a través de Cordova). Después se deben instalar los paquetes necesarios para el nivel de la API para la que se quiera desarrollar la app, en este caso es la API v23 (hasta Android 6.0). Como mínimo, Ionic soporta desde las versiones Android 4.1+ y 4.1.1+.
- Sublime Text 3 build 3083 (editor de texto utilizado para el desarrollo del código)

En cuanto al control de versiones, los cuatro artefactos software se han gestionado a través de Subversion en un repositorio de la empresa Eurohelp (las apps se han subido al final de su desarrollo para que estén en dicho repositorio). Para ello, se han utilizado las extensiones de Eclipse "Subversive - SVN Team Provider v4.0.0" y "Subclipse v1.10.13" obtenidos del *Eclipse Marketplace*.

Por otro lado, las dos aplicaciones se han gestionado también en un repositorio Git personal, para lo cual se ha utilizado Git para Windows v2.7.0 y, aunque al principio se gestionaba mediante el CLI de Git, posteriormente se ha hecho uso de la herramienta SourceTree v1.9.5.0 para hacerlo con una interfaz gráfica.

Tanto los *plugins* de Cordova para el acceso a características de los dispositivos como las librerías JavaScript gestionadas con Bower utilizadas para cada aplicación se detallarán en los apartados correspondientes de cada una en el capítulo 6. En concreto, en los apartados de implementación de las mismas (6.3.3 y 6.4.3).

Creación de diagramas

En el presente documento se muestran algunos diagramas en los capítulos 6 (UML, *Unified Modeling Language*) y 7 (diagramas de Gantt, etc.) que se han realizado con las siguientes herramientas:

- StarUML v2.7.0 (versión de evaluación)
- Microsoft Project Professional 2013 (versión de evaluación)
- Aplicación "Draw.io Diagrams" de Google Drive

Capítulo 6

6. DESARROLLO DEL SOFTWARE

Este capítulo corresponde al análisis de cada uno de los cuatro (4) artefactos software objeto del proyecto de fin de grado. Se estructura en cuatro partes, correspondientes a cada aplicación desarrollada (dos *Building Blocks* en forma de REST API y dos servicios públicos de WeLive o apps para móviles desarrollados de forma híbrida como se ha ido explicando en los capítulos anteriores).

Es importante señalar que, debido a las similitudes en la estructura, arquitectura o funcionalidades que hay entre los *Building Blocks* y las apps, algunas secciones sobre todo hacen referencia al artefacto en el que ya se haya explicado, a fin de evitar volver a analizar innecesariamente las mismas cosas. Además, algunos conceptos y arquitecturas también se han descrito en capítulos anteriores, por lo que se referirá a ellos cuando corresponda.

6.1. UsersFeedbacks BB

En este apartado se procede a analizar, una a una, las fases de desarrollo del *Building Block UsersFeedbacks*. Se estructura en análisis, diseño, implementación, verificación y pruebas y, finalmente, despliegue. Cada sección incluye los correspondientes aspectos que se deben tratar.

6.1.1. **Análisis**

En este apartado se describen los requisitos que el servicio *UsersFeedback* debe tener en base a las necesidades planteadas en el proyecto WeLive.

6.1.1.1. Especificación y requisitos

El BB consiste en un servicio que permita gestionar comentarios o *feedbacks* que en una aplicación se realizan en relación a algún objeto o *ítem*. Debe ser por tanto un servicio genérico que pueda utilizarse por cualquier aplicación móvil de WeLive que necesite incluir la funcionalidad necesaria para que los usuarios de una app puedan comentar y valorar algo (una propuesta, una noticia...) y puedan ver después dichos *feedbacks*.

Un ***feedback*** consiste en un **comentario y en un *rating* del 1 al 5** que un usuario realiza sobre un objeto correspondiente a una aplicación. Se debe guardar también la fecha de creación del mismo.

Requisitos técnicos

En el aspecto técnico, son requisito de la empresa Eurohelp los siguientes aspectos (estas herramientas y tecnologías se han analizado en las secciones 5.3 y 5.4):

- El servicio web debe estar en forma de API REST, utilizando para ello el *framework* Spring con Java y, en concreto, la herramienta Spring Boot.
- Para la gestión de dependencias del proyecto Spring, por otro lado, se debe usar Maven.
- Para la gestión de control de versiones hay que usar Subversion, a través siempre del entorno de desarrollo Eclipse.
- El despliegue del servicio se debe realizar en la plataforma Cloud Foundry con el dominio específico del proyecto WeLive.

Requisitos funcionales

En cuanto a las funcionalidades que el BB debe ofrecer, y mencionadas en la sección 3.1 relativa al alcance del proyecto, se enumeran a continuación:

- Debe ser un **servicio genérico** utilizable por cualquier aplicación de WeLive.

- Un usuario puede **crear un *feedback*** indicando el texto (comentario en sí) y el *rating* (una valoración del 1 al 5), sobre un objeto de la app.
- Se pueden **obtener los *feedbacks*** creados en relación a un objeto de la app. Para ello se pueden indicar, de manera opcional, una serie de parámetros para ver los comentarios filtrados como: obtener los que se hayan creado a partir de una determinada fecha, hasta una fecha, y/o el número de comentarios a obtener. Esto sirve para que la app pueda cargar los comentarios por bloques a modo de paginación, de manera que no tenga que cargar todos en la app de una vez.
- En relación a la paginación comentada en el punto anterior, el servicio también ofrece un método con el que obtener el **número de *feedbacks* nuevos creados a partir de una fecha**, a fin de que la app sepa que hay comentarios nuevos para cargar.
- Se puede obtener la **media de *ratings*** de los *feedbacks* asociados a un objeto, junto con el número de *feedbacks* con los que se ha calculado.
- Por último, ofrece un método para obtener el WADL (*Web Application Description Language*) del servicio en formato XML, de forma que un desarrollador pueda ver los métodos y funcionalidades disponibles y la forma de uso (aunque no los formatos de respuesta).

6.1.1.2. Modelo del Dominio

A diferencia de otro tipo de programas desarrollados por ejemplo en la asignatura de Ingeniería del Software, en este apartado al tratarse de un servicio REST con un controlador que accede a la capa de acceso a datos, lo que se va a mostrar a continuación sobre todo es el modelo para los Objetos de Transferencia de Datos o DTOs (*Data Transfer Object*) utilizados en el servicio.

NOTA

Se denomina DTO *(Data Transfer Object)* a un objeto que transporta datos entre procesos, cuya creación está normalmente motivada por la comunicación entre interfaces remotas como servicios web a fin de encapsular dichos datos en una misma llamada. De este modo se reduce el número de llamadas necesarias para enviar varios datos entre un proceso y otro. Por tanto, simplemente es un objeto que no contiene ninguna lógica de negocio y sirve para almacenar y entregar (transferir, "*transfer*") datos.

***** No confundir con un DAO (*Data Access Object*), que es un componente software que proporciona una capa de acceso a datos de diferentes orígenes (y que también existe en el servicio *UsersFeedbackBB*).

Tabla 4: Definición de DTO (Data Transfer Object)

En la Figura 22 se muestran las entidades utilizadas (DTOs) en respuesta a distintas llamadas al servicio y que, para dichas respuestas, se convierten directamente en un objeto JSON con el mismo formato para su uso por el cliente del servicio. Las primeras dos también se usan como modelo del dominio internamente en el servicio.

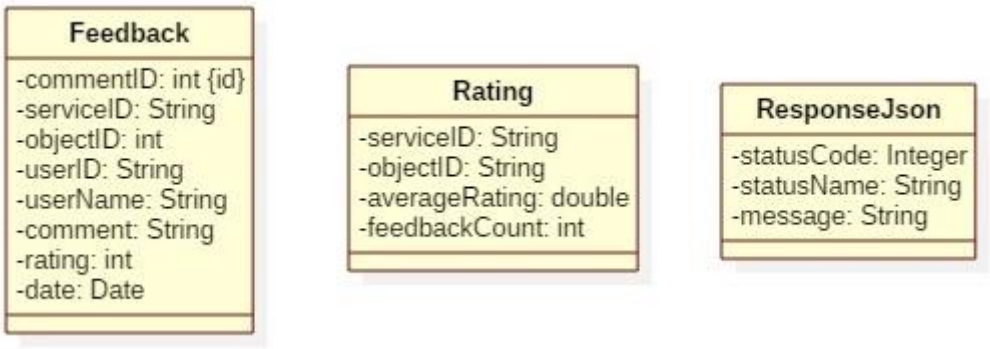


Figura 22: DTOs definidos para UsersFeedbackBB

Por tanto, son tres las entidades definidas para trabajar como DTOs, de manera que las respuestas que la API REST *UsersFeedbackBB* va a devolver en las diferentes llamadas corresponderán a alguna de las tres, en formato JSON. El *mapeo* de objeto Java a JSON se hace automáticamente, como se menciona en la explicación del diagrama se secuencia (6.1.2.4) y de la implementación (6.1.3).

La entidad ***Feedback*** (cuyo rating está en el rango 1-5) se utiliza como respuesta al método de obtener los *feedbacks* de un objeto, la entidad ***Rating*** al de obtener la media del *rating* de los *feedbacks* de un objeto, y la entidad ***ResponseJson*** es una entidad auxiliar que se utiliza para más de un propósito:

- Como respuesta satisfactoria al método de crear un *feedback*.
- Como respuesta al método para obtener el número de *feedbacks* a partir de una determinada fecha.
- Como respuesta a cualquier tipo de error (*feedbacks* no encontrados o cualquier otro error producido, como parámetros ilegales (falta un parámetro, un parámetro es *null*...), algún error de acceso a la base de datos o cualquier otro error producido en tiempo de ejecución).

6.1.2. **Diseño**

En esta segunda sección del apartado correspondiente a *UsersFeedbackBB* se tratan los aspectos correspondientes al diseño del servicio web REST. Se mostrarán los diagramas específicos para el modelo de CU, diagrama de clases y diagrama de secuencia, además de mencionar el formato de los recursos identificados para el servicio además de todo lo concerniente al modelo de datos.

6.1.2.1. Modelo de Casos de Uso

En la siguiente Figura 23 se muestra el diagrama de CU definido para el servicio web. En este caso el actor corresponde a un sistema o proceso externo, el cual hace uso de los CU definidos para esta API. Conviene mencionar que en el modelo no se ha incluido la funcionalidad que permite obtener el WADL (en formato XML) del servicio, al considerarla algo aparte de los CU definidos para el mismo.

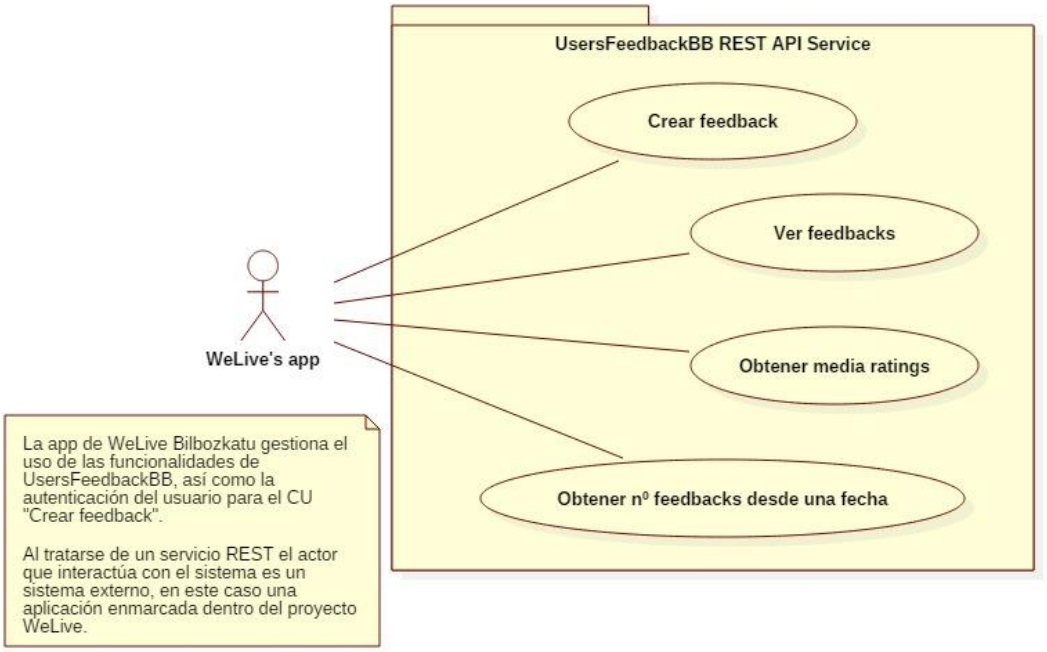


Figura 23: Modelo de Casos de Uso de UsersFeedbackBB

6.1.2.2. Identificación de recursos

Al tratarse de un servicio web REST dispone de varios recursos a los que un cliente, a través del protocolo HTTP, puede acceder. Estos recursos o métodos básicamente son los comentados en los puntos anteriores, pero se deben definir una serie de parámetros a fin de implementarlos en el servicio, en concreto, en el controlador.

Estos parámetros o propiedades son la **URI** de direccionamiento (el controlador redirigirá la petición a la función correspondiente) y el **método HTTP**. Dicho método, o verbo HTTP, especifica la acción a realizar en la petición y el protocolo HTTP define algunos como POST, GET, PUT o DELETE de las operaciones conocidas como CRUD (*Create Read Update Delete*) respectivamente. En este caso no existen operaciones de actualización o eliminación de ningún objeto, por lo que los métodos usados son GET y POST.

Recurso para crear un *feedback* (*addFeedback*)

Propiedad	Valor
URI	/feedback/add
Método HTTP	POST
Descripción	Sirve para crear un *feedback* (comentario y *rating* de 1 a 5) asociado a un usuario y objeto.

Tabla 5: Propiedades REST de addFeedback (UsersFeedbackBB)

Los parámetros deben estar en formato JSON (se envían en el *body* de la petición HTTP). La siguiente Figura 24 muestra cómo los parámetros "serviceID", "objectID", "userID", "userName", "comment" y "rating" deben ser enviados (el *rating* debe ser un número del 1 al 5 aunque sea de tipo String).

```
{
        "serviceID": "bilbozkatu",
        "objectID": "1",
        "userID": "6487",
        "userName": "Mario",
        "comment": "Esto es un comentario",
        "rating": "3"
}
```

Figura 24: Ejemplo JSON de petición para crear un feedback en UsersFeedbackBB

Recurso para obtener *feedbacks* (*getFeedbacks*)

Propiedad	Valor
URI	/feedback/list
Método HTTP	GET
Descripción	Sirve para obtener *feedbacks* en relación a un objeto. Opcionalmente, se pueden indicar los parámetros *resultsFrom*, *resultsTo*, *resultsCount* o una mezcla de ellos para limitar o filtrar los *feedbacks* obtenidos en base a un rango de fechas o número de *feedbacks* a obtener.

Tabla 6: Propiedades REST de getFeedbacks (UsersFeedbackBB)

Los parámetros, además de los indicados en la descripción (*resultsFrom* y *resultsTo* deben cumplir el formato "YYYY-MM-DD hh:mm:ss Z" para fechas, siendo Z la zona horaria en formato "+0100" o "+01"), son "serviceID" y "objectID" para hacer referencia a la app y al objeto del que se pregunta.

Recurso para obtener la media de *ratings* (*getFeedbacksAverageRating*)

Propiedad	**Valor**
URI	/feedback/list/average
Método HTTP	GET
Descripción	Sirve para obtener la media de *ratings* de los *feedbacks* asociados a un objeto, además del número de *feedbacks* con el que se ha calculado dicha media.

Tabla 7: Propiedades REST de getFeedbacksAverageRating (UsersFeedbackBB)

Los parámetros son "serviceID" y "objectID" para hacer referencia a los *feedbacks* de un objeto de una app, al igual que en el recurso anterior.

Recurso para obtener el nº de *feedbacks* a partir de una fecha (*getFeedbacks CountFromDate*)

Propiedad	**Valor**
URI	/feedback/list/count
Método HTTP	GET
Descripción	Sirve para obtener el nº de *feedbacks* almacenados a partir de una determinada fecha hasta la actual (la de la realización de la llamada).

Tabla 8: Propiedades REST de getFeedbacksCountFromDate (UsersFeedbackBB)

Los parámetros son iguales a los del punto anterior, "serviceID" y "objectID", además de "resultsFrom" al estilo del recurso *getFeedbacks*, con el formato "YYYY-MM-DD hh:mm:ss (+ZZZZ|+ZZ)".

Algunos aspectos de la implementación y uso de estos recursos (como los parámetros) se detallarán en el apartado 6.1.3 que trata la implementación del servicio *UsersFeedbackBB*.

6.1.2.3. Diagrama de clases

En la siguiente Figura 25 se muestra el diagrama de clases reducido correspondiente al servicio web REST *UsersFeedbackBB* implementado. En este diagrama se ha excluido la relación que las clases de arranque del servicio tienen con el *framework* Spring y Spring Boot. El diagrama extendido se encuentra adjunto en el **Anexo A**, en el apartado correspondiente al *Building Block UsersFeedback*.

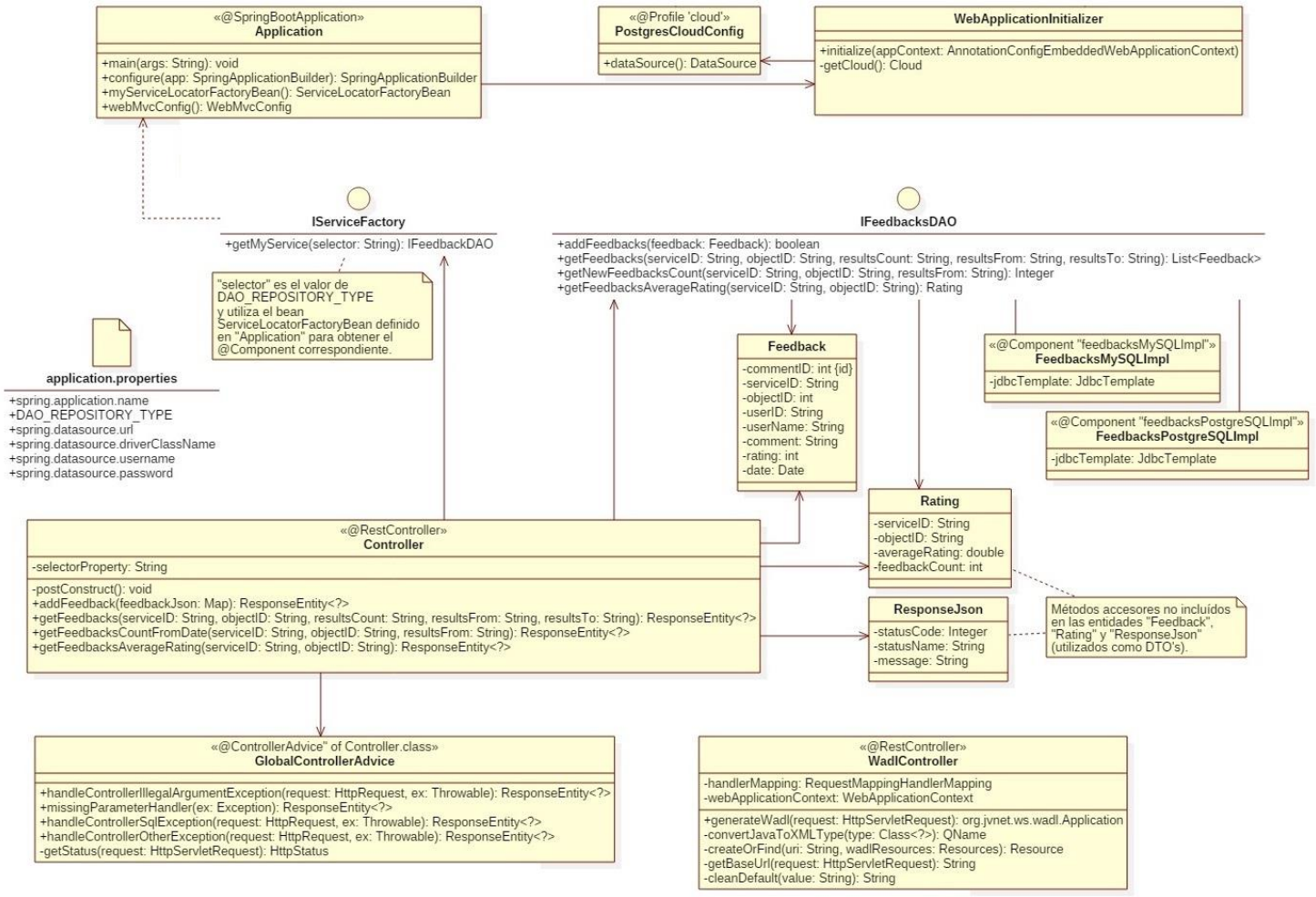


Figura 25: Diagrama de Clases reducido de UsersFeedbackBB

En el diagrama de clases que se muestra en la Figura 25 se puede apreciar a grandes rasgos el modo de trabajar con Spring Boot. Dispone de una clase principal llamada ***Application*** que hace uso de ***WebApplicationInitializer*** para ejecutarse y esta, a su vez, utilizará la clase ***PostgresCloudConfig*** en el caso de que se esté desplegando en la nube (Cloud Foundry lo detectará automáticamente y utilizará el *DataSource* definido en él para conectarse a la base de datos, en este caso configurándolo con los parámetros de acceso (url, *driver*, usuario y contraseña) de la asociación existente en dicha plataforma entre el servicio y la base de datos en la nube). En el caso de que se esté arrancando en local, Spring obtendrá dichos parámetros de acceso a la BD del fichero de configuración externo *application.properties*.

Por otro lado, la clase principal sería ***Controller***, en la que se definen las funciones que se ejecutarán para cada petición definida en la API. Esta clase utiliza, como ya se ha dicho, las entidades ***Feedback***, ***Rating*** y ***ResponseJson*** a modo de DTO para las respuestas a las diferentes peticiones. La clase ***GlobalControllerAdvice*** encapsula los manejadores de los errores que se pueden producir en las funciones del *Controller*, como un error en el acceso a la

BD o cualquier otro error en tiempo de ejecución, gestionándolos y produciendo la respuesta *ResponseJson* correspondiente. ***WadlController*** es un controlador similar a *Controller*, pero su único objetivo es el de generar el WADL del servicio en formato XML, especificando los métodos sus parámetros.

Por último está la capa de acceso a datos. Está representada por la interfaz ***IFeedbacksDAO*** (que evidentemente es un DAO o *Data Access Object*), la cual implementan las clases ***FeedbacksMySQLImpl*** y ***FeedbacksPosgreSQLIml*** para añadir soporte para una base de datos MySQL y PostgreSQL respectivamente. Inicialmente se soportó MySQL, pero posteriormente ha sido necesario el soporte para PostgreSQL debido a que la cuenta de Cloud Foundry que había que utilizar para el despliegue del servicio sólo disponía de una instancia del servicio PostgreSQL en la plataforma.

Al desplegar el servicio se accederá a un soporte de la base de datos u otro en función de la variable DAO_REPOSITORY_TYPE. La clase ***IServiceFactory*** es utilizada por *Controller* para obtener una implementación del DAO u otra en base a dicha variable. Para ello se ha definido un *bean* de Spring del tipo *ServiceLocatorFactoryBean*, lo que otorga a *IServiceFactory* el "poder" para devolver el DAO con el soporte correspondiente, ya que el valor de DAO_REPOSITORY_TYPE será el nombre asignado a cada implementación del DAO. En una ejecución local la variable está definida en *application.properties*, pero en la nube (Cloud Foundry) se debe definir como variable de entorno del servicio en dicha plataforma, es decir, hay que crearla en la nube y especificar su valor (el correspondiente para el soporte PostgreSQL en este caso).

6.1.2.4. Diagrama de secuencia

A continuación, en la Figura 26, se muestra un diagrama de secuencia simplificado del servicio. El flujo de eventos es idéntico para cada petición a la API, con las respectivas lógicas de negocio en cada una (validación de parámetros, sentencias SQL...), por lo que sólo se mostrará el del método de obtener la media de *ratings* de los *feedbacks* asociados a un objeto.

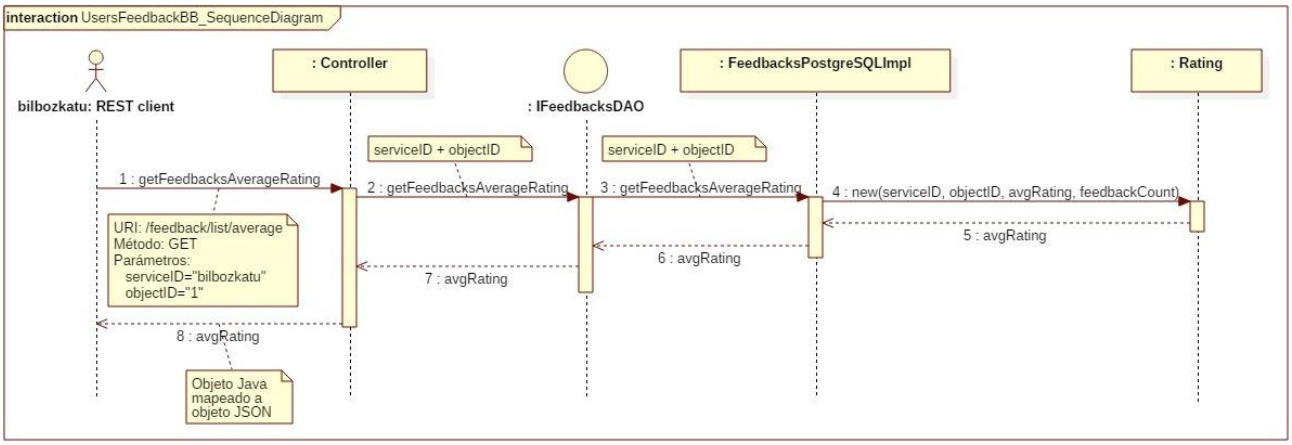


Figura 26: Diagrama de secuencia simplificado de UsersFeedbackBB

En el diagrama se puede ver el funcionamiento general del proceso de una petición REST al servicio. El cliente REST, en este caso la app Bilbozkatu, realiza una petición al recurso de la API que quiere, mediante una llamada HTTP con la URI, método y parámetros correspondientes.

Aunque en el diagrama no se ve, Spring *mapea* la petición a la función del controlador correspondiente en base a la URI especificada, gracias a la anotación *@RequestMapping* que se utiliza en el código del controlador (de *Spring MVC RESTful Web Services*).

La función del controlador que se ha ejecutado obtendrá los parámetros (a veces indicados en la propia URI como parámetros GET ("...?p1=v1&p2=v2") y a veces en el *body* de la petición) y los validará. Si se produce algún problema (falta de parámetros, formato incorrecto, etc.) el controlador enviará una respuesta al cliente REST con el formato de *ResponseJson* antes descrito, con el mensaje de error correspondiente. Si no ha habido ningún problema, el controlador accederá al método correspondiente del DAO (en el diagrama se muestra tanto el interfaz *IFeedbacksDAO* como la implementación *FeedbacksPostgreSQLImpl* para ver su relación, pero realmente no se producen los pasos 3 y 6 del diagrama).

Una vez la implementación correspondiente del DAO haya hecho la operación contra la base de datos, en el ejemplo de la Figura 26 crea un objeto de tipo *Rating* que será el devuelto al cliente REST en formato JSON. Esta conversión de objeto Java a objeto JSON se produce gracias al soporte de conversión de mensajes HTTP de Spring, el cual Spring define que sea "*MappingJackson2 HttpMessageConverter*".

6.1.2.5. Modelo de datos

Una vez analizados los **requerimientos** para el servicio *UsersFeedbackBB*, el modelado de datos ha sido sencillo. El **diseño conceptual** (modelo Entidad-Relación o ER) es tan trivial que no hace falta mostrarlo, ya que consta de una única entidad: *Feedback.*

Esta entidad debe tener los siguientes campos:

- ***commentID***. Identificador del *feedback*.
- ***serviceID***. Identificador de la aplicación.
- ***objectID***. Identificador del objeto que se comenta.
- ***userID***. Identificador del usuario que ha creado el *feedback*.
- ***userName***. Nombre del usuario que ha creado el *feedback*.
- ***comment***. Comentario (texto) del *feedback*.
- ***rating***. Valoración del 1 al 5 del objeto al que se asocia el *feedback*.
- ***date***. Fecha de creación del *feedback*.

En este caso el *commentID* se ha decidido que sea un identificador auto-incrementable de la base de datos. El *serviceID* y *objectID* posibilitarán a la aplicación que utilice el *Building Block* referenciar los comentarios que pertenezcan a un objeto de su app. Al campo *userID* finalmente se le ha añadido el campo *userName* debido a que esta tabla estará en una base de datos independiente a otros BBs de WeLive, y conviene guardar también el nombre del usuario a fin de que la aplicación que lo use pueda mostrar el nombre del usuario que ha realizado el comentario (*userID* se referirá al identificador del usuario obtenido del AAC BB de WeLive, el cual entre los datos del perfil básico del usuario están el *userId* generado). Por último, el campo *date* obtendrá su valor de manera automática en la base de datos al crear el *feedback*.

El siguiente paso en el diseño de las bases de datos consiste en la **elección del SGBD** (Sistema de Gestión de Bases de Datos). En este caso era un requerimiento que la base de datos siguiera el modelo relacional, por lo que inicialmente se optó por una base de datos MySQL. Sin embargo, tal y como ya se ha mencionado, también se ha desarrollado para PostgreSQL debido a que la plataforma Cloud Foundry sólo permitía este tipo de base de datos (cosa que se ha sabido en últimas fases del desarrollo del proyecto).

A continuación se procedería al **diseño lógico**. Sin embargo, dada la simpleza de la entidad diseñada no se procede a ello. En su lugar, se muestra directamente el **diseño físico** de la base de datos, tanto para MySQL como para PostgreSQL.

MySQL

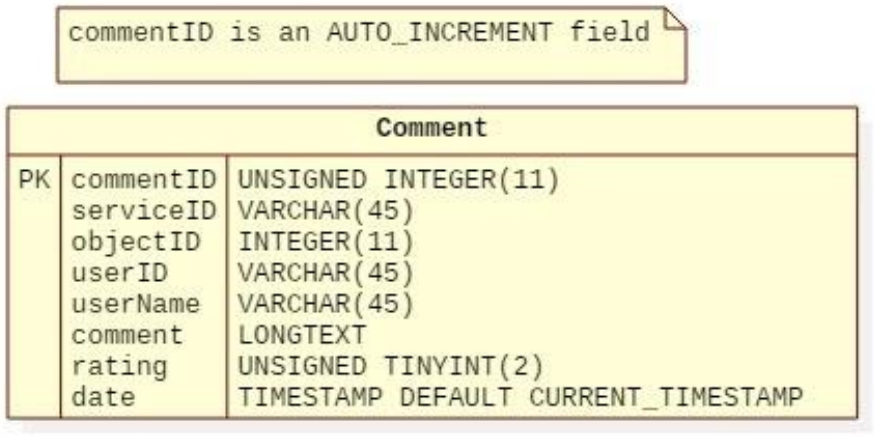


Figura 27: Diseño físico en MySQL para UsersFeedbackBB

La Figura 27 muestra el diseño físico en MySQL realizado para *UsersFeedbackBB.* La estructura no presenta dificultad alguna para entenderla, por lo que no hace falta profundizar en ello. No obstante, cabe añadir que debido a la falta de la propiedad "N" en dicho diseño que indica que un campo en *nullable* o que pueda ser *null*, todos los campos de esta tabla son *not null*, es decir, no puede haber

ninguno sin valor. La fecha y hora, junto con la zona horaria, se genera automáticamente al insertar un *feedback*.

PostgreSQL

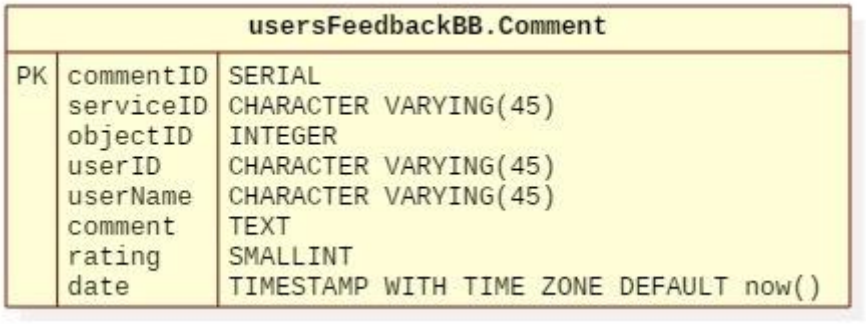


Figura 28: Diseño físico en PostgreSQL para UsersFeedbackBB

En este segundo caso el tipo de los campos se ha seleccionado para que sean equivalentes. Ello y la realización de algunas sentencias SQL ha requerido también de cierta formación, ya que no se había visto específicamente PostgreSQL a lo largo de la carrera. En cuanto al diseño, por ejemplo, el campo *commentID* se ha definido como *serial*, lo que hace que sea un campo al estilo *unsigned integer(11) auto_increment* de MySQL. El campo *date* se ha definido de esa manera para que sea equivalente al de MySQL, como el resto.

Se adjuntan en el **Anexo B** los *scripts* SQL utilizados para la creación del modelo de datos de *UsersFeedbackBB*, tanto para MySQL como para PostgreSQL, incluyendo las restricciones necesarias (*constraints*) para cada uno para que los dos tipos de bases de datos sean equivalentes. Para ello ha sido necesario un cierto aprendizaje de PostgreSQL para realizarlo.

6.1.3. **Implementación**

La implementación de este servicio web ha sido desarrollada, tal y como se ha explicado en el capítulo 4 sobre la metodología utilizada, en una serie de fases incrementales. Si bien algunos cambios en las especificaciones a lo largo del proyecto han hecho que el diseño haya cambiado ligeramente (inicialmente, por ejemplo, no se incluía el *userName* de un usuario), la idea principal ha sido la de obtener soluciones parciales en cada una de las fases.

En esta sección es conveniente recordar al lector algunas de las cosas analizadas en capítulos anteriores. En el apartado 5.3 se han comentado las tecnologías Spring, Spring Boot, la herramienta Maven y la plataforma Cloud Foundry, además del IDE Eclipse en el punto 5.4. Por tanto no se entrará en detalle respecto a dichos *frameworks* o herramientas ya que se ve reflejado en la implementación del servicio tanto en la arquitectura como en algunas técnicas utilizadas.

El siguiente apartado 6.1.3.1 extiende lo ya comentado, explicando brevemente la arquitectura de Spring Boot en relación a Spring, la arquitectura de una aplicación web en Spring y de un servicio web. También se muestra la estructura de ficheros utilizada, alguna técnica empleada y anotación relacionada. Finalmente se añade un apartado con algún problema encontrado.

6.1.3.1. Arquitectura

Como ya se ha ido explicando el objetivo de Spring Boot es el de facilitar y simplificar el proceso de creación de aplicaciones y servicios dentro del *framework* Spring, y obtener así una aplicación Java ejecutable o un *WAR* (*Web application ARchive*, para aplicaciones web) desplegable. En la siguiente Figura 29 se aprecia cómo Spring Boot provee de una capa a los usuarios para el uso del ecosistema Spring y facilitar de este modo su uso.

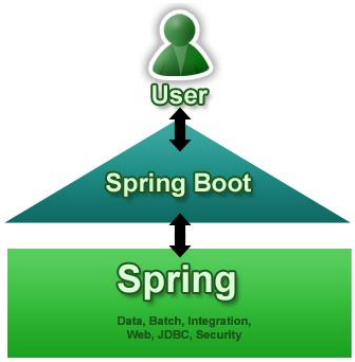


Figura 29: Diagrama de la capa de Spring Boot con respecto a Spring [18]

Extendiendo lo ya comentado en el diagrama de clases y de secuencia en el apartado 6.1.2 en relación al diseño del servicio, a continuación se muestra, en la Figura 30, la arquitectura que una aplicación web Spring puede tener, como es el caso.

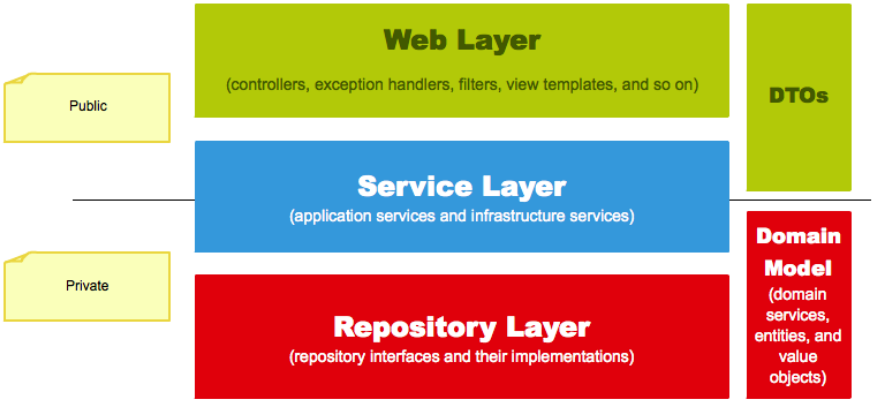


Figura 30: Arquitectura de una aplicación web de Spring [19]

[18] Fuente: https://spring.io
[19] Fuente: http://www.petrikainulainen.net/wp-content/uploads/spring-web-app-architecture.png

La arquitectura mostrada es una arquitectura general para aplicaciones web en Spring que separa en distintas capas las responsabilidades de las mismas, haciendo efectivo el principio de intereses (*Separation of Concerns* o *SoC*). Una vez especificada esta arquitectura se definirían los llamados DTOs y entidades del dominio, aunque en el servicio web implementado se han reutilizado los mismos tal y como se ha explicado en el modelo del dominio en el punto 6.1.1.2.

- La capa *web* es responsable de la interacción con el usuario y de gestionar los errores de otras capas. Aquí se incluirían, por ejemplo, las vistas que una web tiene. En *UsersFeedbackBB* corresponde al controlador del servicio REST.
- La capa de servicio contiene los servicios de aplicación y de infraestructura, los que proveen la API pública y se comunican con recursos externos como bases de datos, respectivamente.
- Finalmente la capa de repositorio es la menor de la aplicación web, la cual se encarga de comunicarse directamente con el Sistema de Gestión de Bases de Datos correspondiente. Es el DAO (*Data Access Object*) implementado en *UsersFeedbackBB.*

REST vs Servicios Web tradicionales (basados en SOAP)

Para terminar con la descripción de la arquitectura utilizada en *UsersFeedbackBB* que se ha ido explicando en fases anteriores y en el punto 5.3, conviene mencionar brevemente qué es REST y por qué ha surgido.

REST (*REpresentational State Transfer*) no es un estándar (aunque esté basado en estándares como HTTP o URI), sino un estilo de arquitectura de software conocido como *RESTful* para sistemas en la web en base a una serie de restricciones. Aplica el foco en los recursos de un sistema, incluyendo el modo de acceder a ellos y la forma de transferirlos, siguiendo cuatro principios: utilizar métodos HTTP, no mantener el estado (cliente-servidor) entre llamadas, exponer URIs con forma de directorios y transferir XML, JSON, o ambos. Estos aspectos se han ido analizando en las fases de análisis y diseño del servicio.

El desarrollo de servicios web utilizando el estilo REST está desbancando a los tradicionales servicios web basados en el protocolo SOAP (*Simple Object Access Protocol*) (no confundir estilo con protocolo). Esto es debido a que la comunicación entre un servicio web SOAP y un cliente está muy restringida en el sentido de que todo puede fallar si cualquiera de las dos partes cambia algo. El mantenimiento es más complicado, a diferencia de un servicio web que no utiliza el protocolo SOAP y aplica el estilo REST. Con este último se convierte en un servicio genérico que sabe cómo usar un protocolo y métodos estandarizados, y cualquier cliente (aplicación, etc.) sólo tiene que "entrar" dentro de ese marco de trabajo.

Estructura de ficheros

En la siguiente Figura 31 se indica la estructura de ficheros implementada de acuerdo al diagrama de clases (punto 6.1.2.3) y la arquitectura utilizada que ya se han analizado. Como se puede ver, en el primer nivel de la estructura se encuentra el paquete ***eus.eurohelp.welive.usersFeedbackBB***, con las clases *Application* y *WebApplicationInitializer* que ponen en marcha el servicio y configuran lo necesario (*beans, datasource,...*).

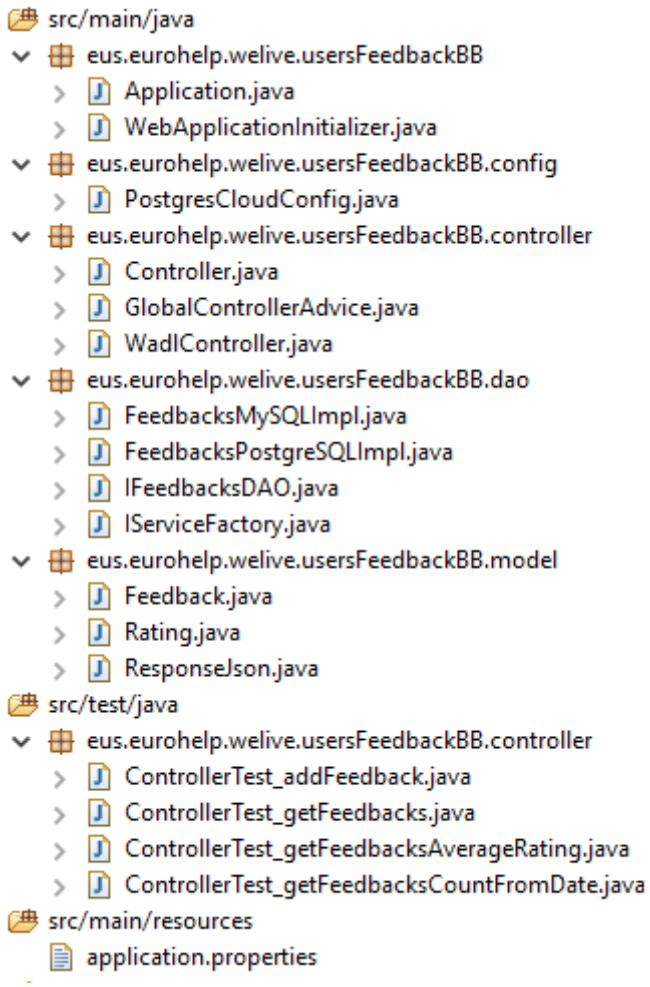


Figura 31: Estructura de ficheros de UsersFeedbackBB

A partir de ahí se estructura en los siguientes paquetes:

- ***config***. Contiene la clase para la configuración del *DataSource* en Cloud Foundry (se utiliza en el despliegue en la nube, no en local).
- ***controller***. Contiene el controlador principal del servicio junto con el *GlobalControllerAdvice* que encapsula los errores que en él se producen y el *WadlController* para ofrecer el *WADL* del servicio en una URI concreta en formato XML.
- ***dao.*** Contiene lo necesario para la capa de acceso a datos. El *IServiceFactory* utilizado en *Controller* para obtener el soporte para

MySQL o para PostgreSQL en función de una variable de configuración (de *application.properties* en local, o definida en la plataforma Cloud Foundry), y la interfaz del DAO y sus implementaciones.

- ***model.*** Contiene las entidades utilizadas como DTOs en el servicio (y como entidades del modelo del dominio entre el controlador y el DAO).

En *src/main/resources* se encuentra el fichero de configuración externo *application.properties*, en el cual se configuran los parámetros de acceso a la base de datos (*datasource* de Spring) y la variable para indicar si se debe usar el soporte para MySQL o PostgreSQL. Estos parámetros son aplicables tanto en una ejecución local como si se desplegara en un servidor, por ejemplo *Tomcat*. Sin embargo se ignoran a la hora de desplegar en la plataforma Cloud Foundry en la nube y, en su lugar, la configuración se realiza con la clase definida en *config,* para lo cual se define un *DataSource* básico que indica que la base de datos a usar es la única que está asociada a *UsersFeedbackBB* en Cloud Foundry (al subir *UsersFeedbackBB* a Cloud Foundry se asocia la app a la instanacia de una base de datos, PostgreSQL en este caso).

Por último, en *src/test/java* se encuentran todos los test unitarios definidos con JUnit, siendo cada fichero el correspondiente para cada recurso disponible en el servicio. Se realizan los test tanto de validación de parámetros como de los posibles resultados.

Técnicas aplicadas

Además del *mapeo* de objeto Java a objeto JSON que se realiza de manera automática (comentado en el punto 6.1.2.4), la aplicación del estilo REST en el servicio o la validación exhaustiva de los parámetros de los diferentes recursos (enviados en la propia URI en algunos casos, o con formato JSON en otros como en *addFeedback,* y los cuales se validan con anotaciones Spring y otras comprobaciones), cabe destacar un par de técnicas aplicadas en la implementación.

La primera corresponde al **Patrón Factoría** de Spring. A pesar de que al inicio del desarrollo no se contemplaba añadir soporte para PostgreSQL, sí se pensó en implementar el DAO de forma que la inserción de soportes para otras bases de datos fuera directa. De este modo, además de implementar la interfaz del DAO (*IFeedbackDAO*) y su implementación *FeedbackMySQLImpl* (lo que sería el *Patrón Object*), se quiso externalizar la configuración del soporte de base de datos a utilizar, en este caso con la variable *DAO_REPOSITORY_TYPE* del fichero de configuración *application.properties*.

Para ello, después de investigarlo, se ha realizado utilizando técnicas de Spring. En concreto, se ha definido un *bean* de tipo *ServiceLocatorFactoryBean* de Spring en *Application* (clase que también sirve para definir *beans*). En la siguiente Figura 32 se muestra la configuración de dicho *bean*, en la que se configura la

interfaz *IserviceFactory* como la "constructora" del DAO en base a una variable externa.

```
@Bean
public ServiceLocatorFactoryBean myFactoryServiceLocatorFactoryBean()
{
    ServiceLocatorFactoryBean bean = new ServiceLocatorFactoryBean();
    bean.setServiceLocatorInterface(IServiceFactory.class);
    return bean;
}
```

Figura 32: Bean de Spring par el Patrón Factoría de UsersFeedbackBB

Una vez definido el *bean* de la Figura 32, a continuación se muestra su uso en el controlador, en donde se accederá al DAO.

```
@RestController
public class Controller {

    @Value("${DAO_REPOSITORY_TYPE}") private String selectorProperty;

    @Autowired private IServiceFactory feedbacksDAOFactory;
    private IFeedbacksDAO feedbacksDAO;

    @PostConstruct
    public void postConstruct()
    {
        this.feedbacksDAO = feedbacksDAOFactory.getMyService(selectorProperty);
    }
```

Figura 33: Atributos para el DAO en el controlador de UsersFeedbackBB

En la Figura 33 se muestran los atributos definidos en el controlador, antes de proceder a implementar las funciones correspondientes a los recursos del servicio. El atributo *selectorProperty* obtendrá, gracias a la anotación *@Value*, el valor de la variable externa *DAO_REPOSITORY_TYPE*. La anotación *@Autowired* de *feedbacksDAOFactory* instanciará la factoría antes configurada (no es más que una interfaz con el método *getMyService* que devuelve el DAO *IFeedbacksDAO*), y se utiliza en el método *@PostConstruct*. Dicho método se ejecuta después de instanciar el controlador (entendido en Spring como un *bean*), y utilizará dicha Factoría *feedbacksDAOFactory* instanciada para obtener la implementación de la interfaz DAO correspondiente a la indicada en la variable *DAO_REPOSITORY_TYPE* (patrón Factoría).

Por último, se debe identificar con un nombre cada implementación que se puede "construir" en dicha factoría, y poder indicar así una u otra mediante la variable mencionada. Para ello las implementaciones del DAO se han configurado tal y como se puede ver en la Figura 34 a continuación. Gracias a la anotación *@Component* se indica el nombre asociado a este componente, "feedbacksPostgreSQLImpl" en este caso para el soporte de PostgreSQL. Por otro

lado, se puede ver que Spring instanciará el objeto *JdbcTemplate* con el *DataSource* que haya tenido (local, en la nube...), y operar así con sentencias SQL (*INSERT, SELECT,...*) directamente en la base de datos.

```
/**
 * Data Access Object implementation for PostgreSQL DataBase.
 *
 * @author ilatorre
 */
@Component("feedbacksPostgreSQLImpl")
@Repository
public class FeedbacksPostgreSQLImpl implements IFeedbacksDAO{

    @Autowired
    private JdbcTemplate jdbcTemplate;
```

Figura 34: Identificación en Spring del soporte PostgreSQL para UsersFeedbackBB

El segundo punto que se quiere destacar es el **estudio y uso de anotaciones de Spring**: *@SpringBootApplication*, *@RequestMapping, @CrossOrigin* (para habilitar las cabeceras o *headers CORS (Cross-Origin Resource Sharing*) en las respuestas del servicio, que es un concepto de seguridad importante implementado para navegadores web para prevenir que el código JavaScript haga peticiones a un origen diferente del que se servía), *@Repository* o *@Component*, entre otros. Estas anotaciones sirven para configurar multitud de cosas en un servicio o aplicación web en Spring.

Además de las propias anotaciones, se ha hecho uso de ***Java Config***, un estilo de configuración que se realiza directamente utilizando Java (para los *beans* en la clase *Application* o la configuración del *DataSource* en la nube de la clase *PostgresCloudConfig*, por ejemplo). Este tipo de configuración difiere con el utilizado en la escasa aproximación que se ha hecho de Spring en la asignatura de Ingeniería del Sofware II, el cual utilizaba ficheros XML para ello.

6.1.3.2. Problemas encontrados

En general el propio uso de un *framework* como Spring y, en concreto, de Spring Boot, ha supuesto un problema en el sentido de que no se había trabajado a lo largo de la carrera. El uso del Patrón Factoría de Spring, por ejemplo, ha tenido cierta complejidad sobre todo a la hora de desplegar en la plataforma Cloud Foundry, ya que después de haberlo conseguido en un equipo local no funcionaba correctamente en la nube. La correcta configuración y uso de algunas cosas han llevado su tiempo de formación y solución de problemas.

El despliegue en la nube ha traído también otros problemas, específicamente por haber cambiado el soporte de MySQL a PostgreSQL. La validación del formato de

los parámetros que representan fechas finalmente se ha implementado con expresiones regulares, de manera que una fecha cumpla el patrón 'YYYY-MM-DD hh:mm:ss (+ZZZZ|+ZZ)'. En este caso, los dos tipos de formato aceptados para la zona horaria son "+0100" y "+01".

En cuanto al soporte para la base de datos PostgreSQL el mayor problema al desplegarlo en Cloud Foundry ha venido del uso de fechas en las consultas SQL. Principalmente han surgido dos problemas al respecto:

1. A la hora de obtener *feedbacks* cuyas fechas de creación deben estar antes y/o después de otra determinada fecha y hora, la **comparación de fechas** de la consulta SQL no se realizaba correctamente. En resumen, después de investigarlo a fondo, esto sucedía porque la precisión de la fecha y hora no era la misma entre la fecha almacenada en la base de datos y la fecha que se indicaba como parámetro en la consulta SQL.

   En la app Bilbozkatu los *feedbacks* que han enviado los usuarios de una propuesta se visualizan por bloques de más nuevo a más viejo, de manera que según el usuario va hacia abajo en la pantalla se van cargando nuevos bloques de *feedbacks* a modo de paginación. Sin embargo, puede suceder que después de haber cargado el primer bloque de *feedbacks* otro usuario haya creado uno más nuevo. En Bilbozkatu, para ello, se realiza una consulta periódica para saber si hay *feedbacks* más nuevos, y si es así obtenerlos. En ese caso se coge la fecha del *feedback* más nuevo que tiene, y le pide a *UsersFeedbackBB* más *feedbacks* a partir de esa determinada fecha.

   En este punto, el problema sucede ya que la app ha obtenido la fecha de los *feedbacks* con precisión de segundos, pero en la base de datos están con milisegundos. Por ejemplo, si la app tiene el comentario más nuevo a las 17:32:57, la consulta **obtendrá dicho *feedback* de nuevo** porque lo tiene guardado como 17:32:57.345, siendo por tanto posterior a la fecha y hora indicada como parámetro en la consulta con precisión de segundos (17:32:54.000).

   Para solucionarlo en la consulta al comparar con la fecha almacenada en la base de datos ésta se **fuerza a redondearse a segundos**, teniendo las dos fechas de la comparación con la **misma precisión**. Esto también ha tenido su complejidad, pero finalmente se ha conseguido con la función *date_trunc('seconds', date)* de PostgreSQL.

2. El segundo problema sólo se ha podido detectar al probar la aplicación Bilbozkatu con *UsersFeedbackBB* desplegado en la nube. Por no entrar en detalle, decir que el problema se ha debido también a comparaciones de fechas, pero por otra razón. Al haber configurado el campo de fecha de un *feedback* como un campo cuyo valor se genera al insertar un *feedback* (indicando como valor por defecto "*now()*"), y al ser un campo que

guarda no sólo la fecha y hora, sino la zona horaria, el problema se ha debido a que en este caso la **zona horaria** que se almacenaba era la del **servidor donde está la base de datos PostgreSQL en la nube**, distinta a la de Donostia. Las consultas se realizaban teniendo en cuenta sólo la fecha y hora, y no la zona horaria.

El servicio *UsersFeedbackBB* está configurado para que la zona horaria de la fecha y hora de un *feedback* sea la zona horaria de "Europa/Madrid", que puede ser UTC/GMT +01 o +02 en función de si es horario de verano o de invierno. Sin embargo, al realizar la consulta sólo se indica la fecha y hora, aunque esté en la zona horaria de Donostia, pero al comparar en la base de datos se realizaba entre fechas con zonas horarias distintas.

Para solucionarlo, además de especificar la zona horaria en la petición a *UsersFeedbackBB*, ha sido necesario formatear el parámetro de fecha de la consulta SQL para que la tenga en cuenta, de este modo: "*TO_TIMESTAMP("parámetro_fecha", 'YYYY-MM-DD HH24:MI:SS') AT TIME ZONE 'UTC+ZZ'"*, siendo *ZZ* la zona horaria, y "*parámetro_fecha"* la fecha con la que debe comparar que corresponderá a la indicada por la app.

Por tanto, además de otros problemas y dificultades, el tratamiento de fechas ha sido complejo, en concreto, el añadir soporte para PostgreSQL y desplegarlo en la nube. Todo esto, por supuesto, junto con lo ya mencionado sobre Spring y Spring Boot.

6.1.4. **Verificación y pruebas**

En este penúltimo apartado correspondiente al servicio web REST *UsersFeedbackBB* se comentan a grandes rasgos las pruebas realizadas para validar el correcto funcionamiento de la implementación desarrollada, en base al análisis y diseño realizado.

Tal y como ya se ha comentado brevemente en el apartado que habla de la estructura del servicio en el punto 6.1.3.1 (recordar la Figura 31 de la estructura de ficheros), se han realizado una serie de pruebas unitarias a través de la herramienta **JUnit**. Estas pruebas, ejecutables de manera independiente y que no alteran el estado de la base de datos, se agrupan en ficheros distintos en el paquete ***eus.eurohelp.welive.usersFeedbackBB.controller*** del directorio *src/test/java*. Por tanto, son pruebas de cada uno de los recursos o métodos disponibles públicamente en el controlador, siendo cada fichero de pruebas el correspondiente a cada uno de dichos métodos.

Tras una ejecución total de la herramienta JUnit mencionada, a continuación en la Figura 35 se muestra el resultado satisfactorio de las pruebas realizadas para cada recurso del servicio web, tanto a nivel de validación de parámetros como de obtención de diferentes tipos de respuestas.

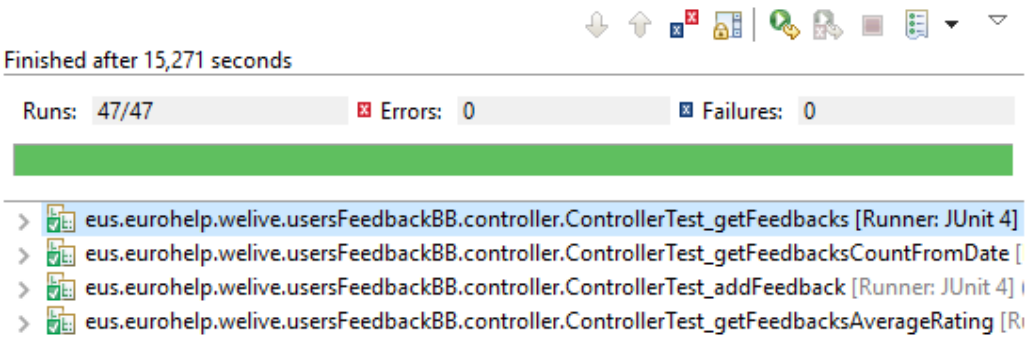


Figura 35: Resultados JUnit de UsersFeedbackBB

En la siguiente Figura 36, en cambio, se indican los resultados detallados de uno de los recursos del servicio a modo de ejemplo. Se trata de *getFeedbacks*, el método con el que obtener los *feedbacks* para un servicio (aplicación) y objeto en concreto ya que es un servicio genérico, y pudiendo indicar algunos parámetros opcionales para poder implementar paginaciones de los *feedbacks* en las aplicaciones obteniéndolos por bloques.

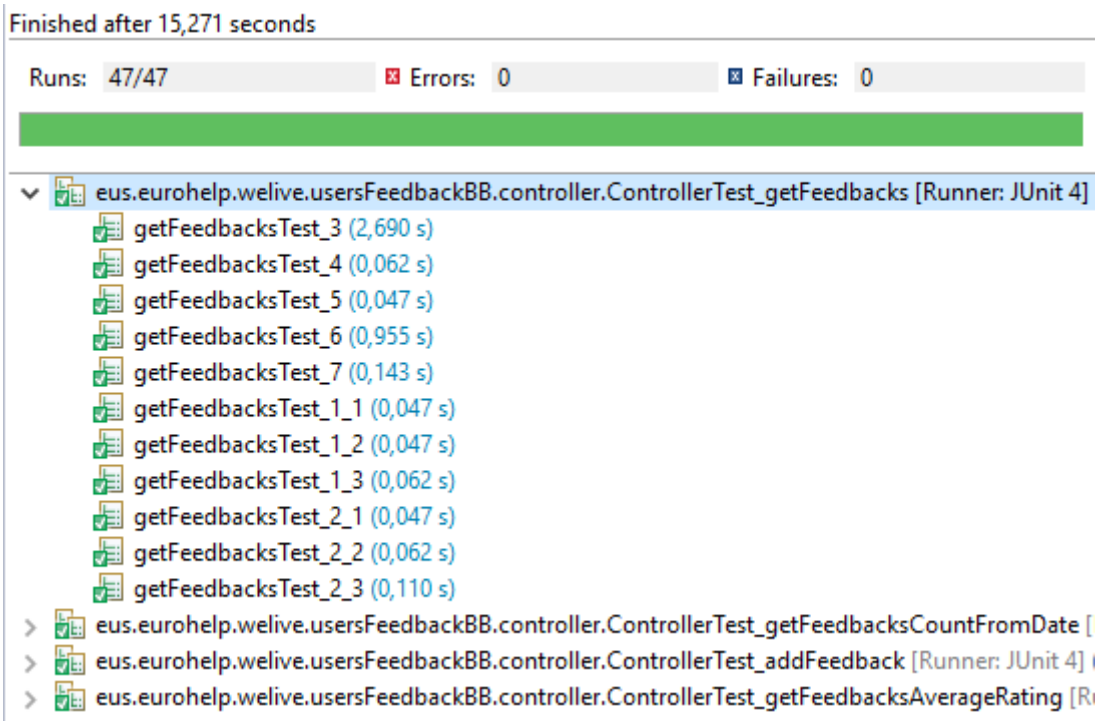


Figura 36: Resultados detallados de JUnit para getFeedbacks

Cabe añadir que algunos test unitarios del recurso se identifican de forma especial de manera que los test de validación de un parámetro en concreto del método (existencia, formato, etc.) puedan agruparse. Así, el sufijo "_X_Y" tiene X como el identificador del parámetro a validar e Y como un identificador secundario dentro de dicho grupo. El resto de test siguen una numeración normal, como se puede ver en la Figura 36.

6.1.5. **Despliegue (Cloud Foundry)**

El despliegue del servicio se ha realizado finalmente en la plataforma Cloud Foundry. Dicha plataforma ya se ha explicado a grandes rasgos en el apartado 5.3 del capítulo de Herramientas y Tecnologías, por lo que en esta sección no es necesario analizarlo en profundidad. No obstante, es importante señalar que el uso de dicha plataforma ha tenido dificultades, sobre todo por haber tenido que colaborar con la empresa FBK de Italia.

Y es que el uso de Cloud Foundry se ha realizado a través de una cuenta específica de WeLive, que tiene su propio *endpoint* de cara a iniciar sesión y hacer uso de sus servicios. Esto ha provocado distintos problemas a la hora de desplegar el servicio *UsersFeedbackBB* en la nube, que finalmente han resultado ser la consecuencia de problemas ajenos al autor del presente proyecto. Es decir, era la empresa FBK la que tenía que solucionar problemas de autenticación, entre otros, para poder hacer un uso correcto de la plataforma.

La inexperiencia en el uso de Cloud Foundry o plataformas similares ha hecho que estos problemas se pensaran que no fueran de FBK, perdiendo así tiempo en tratar de solucionarlo. Finalmente se ha conseguido realizar el despliegue del servicio bajo el dominio ***https://{nombre-servicio}.cloudfoundry.welive.eu***. A grandes rasgos y después de haber solucionado varios problemas, los pasos para conseguirlo han sido estos:

1. A través de la herramienta Maven de Eclipse, **generar el fichero .war** del servicio a desplegar. Esto se consigue configurando un nuevo *Run Configuration* para Maven con los parámetros necesarios, como se muestra en la siguiente Figura 37.

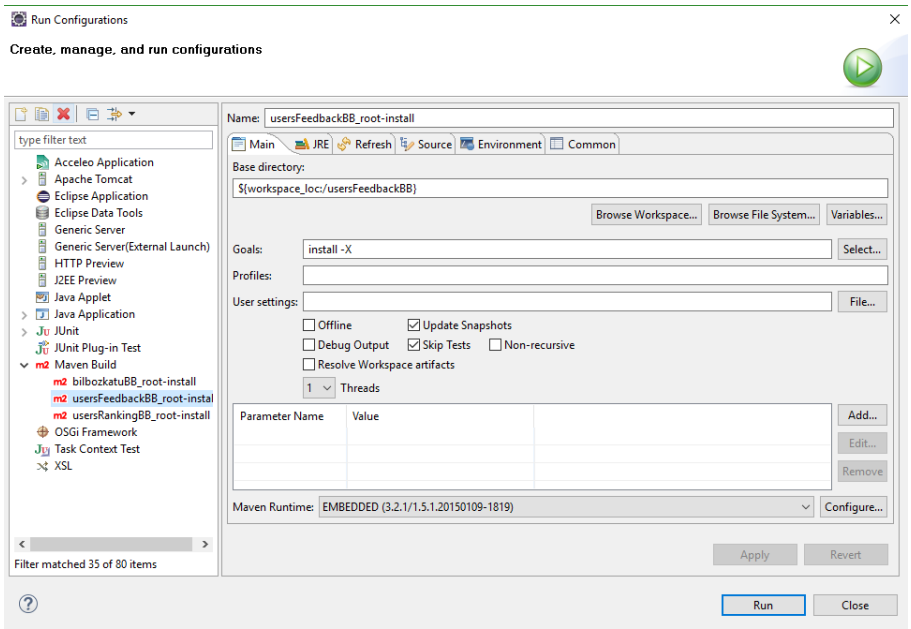


Figura 37: Configuración Maven para generar .war de UsersFeedbackBB

2. A través del CLI de Cloud Foundry, utilizar los comandos ***cf login*** y ***cf push*** para autenticarse y **subir el .war generado**, indicando en cada uno los parámetros necesarios (el *endpoint*, usuario y contraseña en el primero, y el nombre de la app y el *path* o ruta del .war en el segundo).
3. **Instanciar una base de datos PostgreSQL** en la plataforma CloudFoundry. Para ello se debe autenticar en la plataforma y crear dicha instancia de uno de los servicios de bases de datos disponibles para la cuenta utilizada de WeLive. Se puede hacer tanto con el CLI como en la consola web habilitada para Cloud Foundry.
4. **Asociar la aplicación subida a la plataforma con la base de datos instanciada**, de manera que automáticamente acceda al *DataSource* correcto al ejecutar la aplicación (recordar el fichero config/PostgresCloudConfig.java creado para ello, en el que se define un *bean* que acceda a la base de datos asociada a la app, tal y como se ha comentado en el apartado 6.1.3).
5. **Acceder** con un cliente para la base de datos PostgreSQL (en este caso, pgAdmin III) **a la base de datos** instanciada y **crear el modelo físico** mediante *scripts*. Para ello, hay que obtener las credenciales de acceso de la plataforma (url, nombre de la base de datos, usuario y contraseña) que se generan al asociar la aplicación con una base de datos. Estas **credenciales** se encuentran en una variable del entorno de la aplicación en la nube, llamada **VCAP_SERVICES**. En ella se almacenan los datos del servicio PostgreSQL asociado a la aplicación, entre otros las credenciales con el formato siguiente:

```
"credentials":{
        "uri":"postgres://_username_:_password_@_hostname_:_port
                /_database_"
}
```

Figura 38: Formato de credenciales de BD de VCAP_SERVICES en Cloud Foundry

6. Por último, **ejecutar** la aplicación con ***cf start usersFeedbackBB***.

Para concluir con este apartado, cabe destacar que han surgido otros problemas al realizar estos pasos, como que la memoria reservada por defecto para la ejecución de la aplicación inicialmente era insuficiente y ha habido que aumentar el rango, siempre dentro de los límites de la cuenta utilizada. También ha habido dificultades al desplegar más de un servicio a la vez y al parecer sólo había que volver a arrancarlos hasta que lo hicieran, aunque primero se había invertido tiempo por ejemplo en analizar problemas en el código.

6.2. Bilbozkatu BB

A continuación se procede a explicar las fases del desarrollo del *Building Block BilbozkatuBB*. Este artefacto software surge de la necesidad de gestionar los datos de la aplicación móvil Bilbozkatu, independientemente de los comentarios que en ella se gestionan (para lo que se ha usado *UsersFeedbackBB* como ya se ha comentado).

La realización de las diferentes fases de este servicio ha sido la misma que el analizado en el punto 6.1 de *UsersFeedbackBB*. Las etapas y, en concreto, las tecnologías, herramientas, arquitectura Spring y uso de REST, SGBD, técnicas, modo de verificación y plataforma de despliegue son las mismas que las analizadas y descritas en dicho punto 6.1. Por tanto, esta sección se centra en mencionar dichas fases para *BilbozkatuBB* pero sin entrar en detalles ni profundizar en lo ya comentado, tan sólo explicando lo específico de este servicio web.

6.2.1. **Análisis**

En análisis inicial realizado para la aplicación Bilbozkatu en cuanto a funcionalidades a integrar provocó la realización de *BilbozkatuBB*, un servicio web a través del cual gestionar los datos específicos de esta aplicación. Finalmente las necesidades a cubrir por este servicio se han definido de la siguiente manera.

6.2.1.1. Especificación y requisitos

El BB consiste en un servicio web que permita gestionar las propuestas de la aplicación Bilbozkatu (analizada en la siguiente sección 6.3). Dichas propuestas, creadas por los ciudadanos de Bilbao, proporcionan una información valiosa al Ayto. de Bilbao de cara a conocer las preocupaciones o sugerencias de los ciudadanos con el objetivo de mejorar la ciudad. Y es que además de la propia aportación que pueda hacer la gente, un usuario tiene la posibilidad de votar una propuesta a favor o en contra, para tener ese tipo de opinión o valoración de los usuarios además del que ofrece el servicio *UsersFeedbackBB* para enviar comentarios.

Una **propuesta** incluye tanto el título como la descripción de la misma, además de la zona, categoría o tipo (en este caso sólo "ciudadano", pero queda abierta la posibilidad de que el propio Ayuntamiento pueda enviar propuestas a los ciudadanos en un futuro) que están gestionadas por la app que utiliza este *Building Block*. También interesa guardar la fecha de creación de una propuesta, a fin de calcular en la app la caducidad de la misma para permitir, o no, el envío de votos por parte de los usuarios (además de los comentarios con *UsersFeedbackBB*).

Requisitos técnicos

Los requisitos técnicos, impuestos por la empresa, son los mismos que los mencionados en la sección 6.2.1.1 para *UsersFeedbackBB*. Lo aprendido en el uso del *framework* Spring o la plataforma de despliegue del servicio en la nube, entre otros, han servido para la realización de este servicio y para la reutilización de algunos módulos y técnicas empleadas anteriormente.

Requisitos funcionales

Las necesidades y funcionalidades a cubrir en este servicio web son las siguientes:

- Es un **servicio web** en forma de **API REST** para la aplicación móvil Bilbozkatu.
- Un usuario puede **crear una propuesta** asociada a él mismo, indicando el título, descripción, una zona (barrio) de Bilbao y una categoría. Tal y como se ha comentado, estos dos últimos parámetros son gestionados y configurados en la propia app, de modo que el servicio *BilbozkatuBB* no debe realizar ninguna validación al respecto. En este caso también se almacena el tipo ('ciudadano', pudiendo ser otro en el futuro) y la fecha.
- Es posible **obtener las propuestas** creadas, pudiendo hacerlo con datos generales o detallados. Es decir, en la app se muestra una lista con la previsualización de algunas propuestas (mostrando sólo algunos datos de las mismas, datos generales) y también una pantalla con los detalles (incluyendo así todos los datos almacenados). Las opciones a la hora de obtener las propuestas, por tanto, son:
    - Obtener las **propuestas con datos generales** para una pre-visualización de las mismas. Opcionalmente se pueden indicar algunos parámetros para obtener sólo las 'x' propuestas creadas después de alguna de ellas, para ofrecer la posibilidad de obtenerlas por bloques de más nuevas a más viejas.
    - Misma opción que el punto anterior, pero en este caso se pueden indicar también otros parámetros opcionales para **filtrar** las propuestas por **zona, categoría y/o texto** (texto contenido en algunos campos como título o descripción), o un combinación de ellos.
    - Obtener los **detalles de una propuesta**. Opcionalmente, y de cara a que en la app Bilbozkatu en la pantalla de detalles se muestran otros datos como los votos que tiene, se puede indicar el id de un **usuario** para obtener, además de dichos detalles, si ese usuario ya ha votado la propuesta y en tal caso el sentido de su voto.

- El servicio ofrece otra funcionalidad para obtener el **número de propuestas creadas por zona** (de cara a utilizarlo en un mapa que se visualiza en la app, y que se describirá en el apartado 6.3).
- Un **usuario puede votar una propuesta** a favor o en contra. Sólo se permite un voto por usuario y propuesta.
- De cara a controlar los usuarios que han creado o votado las propuestas, es posible **registrar un usuario** en el *Building Block*. Este punto es un tanto controvertido, ya que los usuarios en sí se gestionan, como se ha venido explicando, a través del AAC *Building Block* de Welive. Es decir, a través de ese componente de la plataforma WeLive los usuarios pueden registrarse e iniciar sesión. Sin embargo, la necesidad de controlar el número de votos emitidos por persona, o el nombre de los usuarios que ha creado una propuesta, hace que este servicio requiera de una funcionalidad para registrar el **identificador de usuario generado por el AAC** en el modelo de *BilbozkatuBB*.

  Así, cuando un usuario de Bilbozkatu se registra a través del AAC de WeLive, automáticamente y de manera interna el identificador generado se registra en *BilbozkatuBB*, además de su nombre.
- Al igual que en *UsersFeedbackBB* este servicio ofrece un método para visualizar el WADL (*Web Application Description Language*) en formato XML. De este modo se publican los métodos disponibles, la forma de acceder a ellos y los parámetros que tienen.

6.2.1.2. Modelo del Dominio

En esta sección es aplicable de nuevo lo comentado en el mismo apartado para *UsersFeedbackBB* en el punto 6.1.1.2. Dicho eso, las entidades utilizadas en el servicio como DTOs para responder a las llamadas con datos estructurados (aunque utilizados también como modelo del dominio dentro de las capas software del BB) son las mostradas en la siguiente Figura 39.

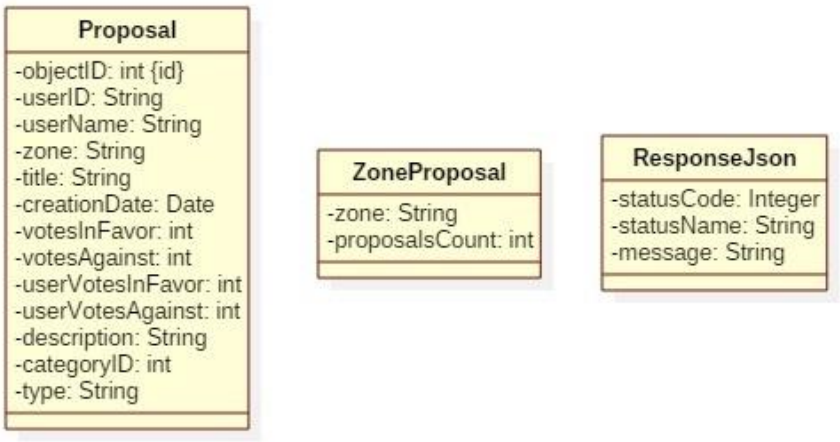


Figura 39: DTOs definidos para BilbozkatuBB

De este modo la estructura indicada para las tres entidades será la que la API de *BilbozkatuBB* usará como respuesta a las diferentes llamadas, en formato JSON. En concreto, la entidad ***Proposal*** se utiliza para el formato de una propuesta al obtener la lista de las mismas o el detalle de una, haciendo la siguiente distinción:

- Si se trata de obtener una lista de propuestas los datos genéricos a devolver son *objectID, zone, title*, *creationDate, votesInFavor* y *votesAgainst*. Estos se mostrarán en la previsualización de las propuestas, aunque la fecha se utiliza para calcular si ya ha caducado o no, y mostrar así el estado "abierta" o "cerrada" en relación a la posibilidad de comentarla o votarla.
- Si se trata de obtener los detalles los datos devueltos son, además de los del punto anterior, el *userID* y *userName* del usuario que creó la propuesta, *categoryID, type* y, si opcionalmente se ha indicado el identificador de un usuario, en *userVotesInFavor* y *userVotesAgainst* se almacena el voto con respecto a esa propuesta de ese usuario, para que se muestre el voto realizado.

La entidad ***ZoneProposal*** sólo se utiliza con el método para obtener el número de propuestas por zona. De este modo, por cada zona existente en la base de datos se asocia esa cantidad. Y, finalmente, ***ResponseJson*** es una entidad auxiliar con la misma función que la detallada en el apartado 6.1.1.2 de *UsersFeedbackBB*: respuesta satisfactoria de algunos métodos o como respuesta informativa de cualquier tipo de error producido.

6.2.2. **Diseño**

Al igual que en el apartado homónimo de *UsersFeedbackBB*, a continuación se analizan las fases de diseño de este servicio *BilbozkatuBB* específico de la app Bilbozkatu. Las explicaciones relativas a algunas cosas como el estilo o identificación de recursos en un servicio web REST, o el diseño de clases y flujo entre las mismas son iguales a *UsersFeedbackBB*, a continuación principalmente se muestran los diagramas necesarios y algunos comentarios necesarios, pero sin entrar al detalle de nuevo.

6.2.2.1. Modelo de Casos de Uso

En la siguiente Figura 40 se muestra el modelo de CU definido para *BilbozkatuBB*. El actor vuelve a ser un cliente del servicio REST aunque, a diferencia de *UsersFeedbackBB* que se trata de un BB genérico, en este caso es únicamente Bilbozkatu la app móvil que lo usa. De nuevo se excluye del diagrama la funcionalidad para obtener el WADL del servicio, aunque presenta una pequeña peculiaridad. Y es que el CU "Ver propuestas por criterios" puede ser indirectamente ejecutado por "Ver propuestas" bajo la condición de que el cliente no haya especificado ningún criterio de búsqueda o filtro en el CU "Ver propuestas".

Si bien esta extensión del CU "Ver propuestas" está tenida en cuenta tanto en el modelo de CU siguiente como en la propia implementación realizada, una vez terminada la aplicación Bilbozkatu esta comprobación también está hecha en la app, de manera que en principio "Ver propuestas por criterios" sólo es invocada por el cliente del servicio y no por el CU "Ver propuestas".

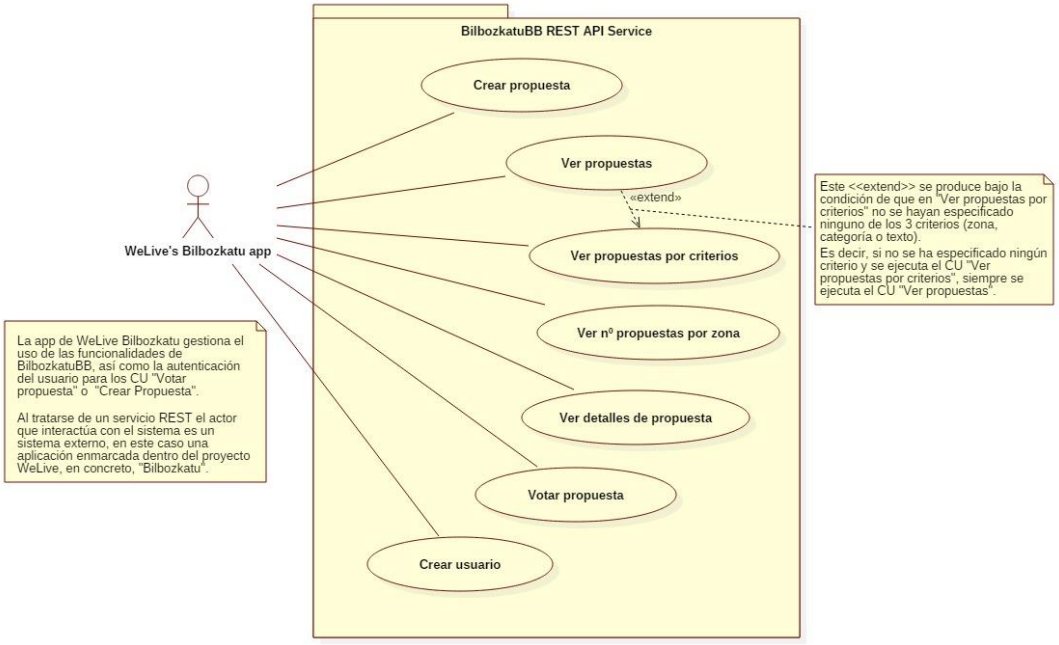


Figura 40: Modelo de Casos de Uso de BilbozkatuBB

6.2.2.2. Identificación de recursos

Seguidamente se indican los recursos del servicio web en forma de REST API de *BilbozkatuBB*, indicando para ellos la URI utilizada, el método HTTP configurado y una breve descripción de los mismos.

Recurso para obtener propuestas (*getProposals*)

Propiedad	**Valor**
URI	/proposal/list
Método HTTP	GET
Descripción	Sirve para obtener los datos generales de las propuestas. Opcionalmente tiene los parámetros *resultsFrom* y *resultsCount* para limitar las propuestas obtenidas.

Tabla 9: Propiedades REST de getProposals (BilbozkatuBB)

Los parámetros opcionales indicados en la descripción de la Tabla 9 sirven para la paginación de propuestas, es decir, para ofrecer a la app Bilbozkatu la

posibilidad de mostrar las propuestas por bloques de más nuevas a más viejas (tal y como se detallará en el apartado 6.3.3 de la implementación de la app Bilbozkatu). No obstante, a diferencia de en *UsersFeedbackBB*, el parámetro *resultsFrom* no es una fecha con la que limitar la paginación, sino el *id* de una propuesta. Así, desde la app se obtienen propuestas más viejas a partir del id de la última propuesta que tiene almacenada.

En este recurso los parámetros se envían en la propia URI de la llamada como parámetros GET, como por ejemplo "*.../proposal/list?resultsFrom=5 &resultsCount=5*". Por supuesto se validan algunas cosas como que *resultsCount* debe ser un valor numérico positivo.

Recurso para obtener propuestas filtradas (*getSearchedProposals*)

Propiedad	**Valor**
URI	/proposal/list/search
Método HTTP	GET
Descripción	Sirve para obtener los datos generales de las propuestas. Opcionalmente tiene los parámetros *resultsFrom* y *resultsCount* para limitar las propuestas obtenidas, pero además dispone de los parámetros *zone, category* y *text* para filtrarlas.

Tabla 10: Propiedades REST de getSearchedProposals (BilbozkatuBB)

Este recurso es idéntico al anterior, con la diferencia de que se pueden filtrar las propuestas en base a la zona, categoría y/o texto. También se indican como parámetros GET y la propia URI del recurso.

Recurso para obtener detalles de una propuesta (*getProposalDetails*)

Propiedad	**Valor**
URI	/proposal/details
Método HTTP	GET
Descripción	Este recurso devuelve todos los datos almacenados de una propuesta en concreto mediante el parámetro *objectID*. Si se especifica el parámetro *userID* también devuelve el voto realizado por ese usuario para esa propuesta, que puede ser que no la ha votado aún, voto a favor o voto en contra.

Tabla 11: Propiedades REST de getPsoposalDetails (BilbozkatuBB)

Los parámetros vuelven a ser de tipo GET, y *objectID* debe ser un valor numérico correspondiente al *id* de una propuesta.

Recurso para obtener el nº de propuestas por zona (*getProposalsCountByZone*)

Propiedad	Valor
URI	/proposal/zones/count
Método HTTP	GET
Descripción	Este método devuelve el nº de propuestas por cada zona existente en la base de datos. Es decir, se tienen en cuenta sólo las zonas utilizadas por las propuestas creadas.

Tabla 12: Propiedades REST de getProposalsCountByZone (BilbozkatuBB)

Este recurso no tiene ningún parámetro, de modo que el uso y acceso a este recurso es más sencillo y no requiere de ninguna otra gestión.

Recurso para votar una propuesta (*voteProposal*)

Propiedad	Valor
URI	/proposal/vote
Método HTTP	POST
Descripción	Sirve para que un usuario genere un voto, a favor o en contra, relativo a una propuesta.

Tabla 13: Propiedades REST de voteProposal (BilbozkatuBB)

En este caso el envío de parámetros es distinto, ya que se realiza mediante un JSON en el cuerpo o *body* de la petición HTTP tal y como se muestra en la siguiente Figura 41. De este modo un usuario vota una propuesta indicando si su votos es favorable o no.

```
{
        "objectID": "4",
        "userID": "187",
        "isFavorable": true
}
```

Figura 41: Ejemplo JSON de petición para votar en BilbozkatuBB

Este tipo de envío de parámetros, en el cuerpo de la petición HTTP, es el modo usual de implementar llamadas a servicios web. Y es que el envío de parámetros en la propia URL tiene menos seguridad, y por tanto se debe evitar en la medida de lo posible en métodos para almacenar datos en alguna base de datos, por ejemplo.

Recurso para crear una propuesta (*addProposal*)

Propiedad	**Valor**
URI	/proposal/add
Método HTTP	POST
Descripción	Sirve para crear una propuesta asociada a un usuario en concreto.

Figura 42: Propiedades REST de addProposal (BilbozkatuBB)

En este caso los parámetros vuelven a pasarse en formato JSON en el cuerpo de la petición HTTP, al igual que en el punto anterior. En la siguiente Figura 42 se muestra el formato de dicho JSON (no se incluye la fecha ya que es un dato generado automáticamente en la base de datos por defecto).

```
{
        "title": "Iluminación en la zona de Abando",
        "userID": "187",
        "zone": "Abando",
        "category": "2",
        "description": "Propuesta para que mejoren la iluminación
                        del barrio de Abando ya que hay numerosas
                        farolas con las bombillas rotas.",
        "type": "ciudadano"
}
```

Figura 43: Ejemplo JSON de petición para crear una propuesta en BilbozkatuBB

Recurso para crear registrar un usuario (registerUser)

Propiedad	**Valor**
URI	/user/add
Método HTTP	POST
Descripción	Sirve para registrar un usuario en el servicio *BilbozkatuBB* para controlar qué usuarios han creado las propuestas y cuántas veces las han votado.

Este recurso se ha añadido posteriormente a lo largo del desarrollo del proyecto debido a que inicialmente se desconocía la forma de autenticarse un usuario en la plataforma de WeLive. Si bien se había tenido en cuenta la necesidad de

almacenar los usuarios, inicialmente estos se almacenaban manualmente en la base de datos. Una vez que se había publicado definitivamente el componente AAC de la plataforma WeLive para la autenticación de usuarios, dicho servicio generaba automáticamente un identificador o *id* de usuario. Esto ha provocado la necesidad de automatizar el registro de usuarios y evitar hacerlo manualmente contra la base de datos.

Por tanto, este recurso requiere de los parámetros *userID* y *userName* del mismo modo del punto anterior, con formato JSON en el cuerpo de la petición HTTP del recurso, de la forma en la que se ilustra en la Figura 44. Cabe añadir que esta funcionalidad es requerida por la app Bilbozkatu una vez el usuario se haya registrado y/o autenticado en la plataforma WeLive, de manera que el *id* generado (y que no varía desde el registro) pueda ser registrado en *BilbozkatuBB* y controlar las propuestas y votos del usuario.

```
{
        "userID": "187",
        "userName": "Patxi",
}
```

Figura 44: Ejemplo JSON de petición para registrar un usuario en BilbozkatuBB

6.2.2.3. Diagrama de clases

El diagrama de clases es prácticamente idéntico al utilizado en *UsersFeedbackBB* (ver el punto 6.1.2.3). Varía en las entidades utilizadas como DTOs excepto para *ResponseJson*, además de los propios métodos del *Controller* y de la capa de acceso a datos (DAO), evidentemente. Los métodos del controlador corresponden nuevamente a los recursos que el servicio web ofrece en forma REST API, y que en este caso son utilizados exclusivamente por la app Bilbozkatu.

Cada uno de estos métodos del controlador accede a la correspondiente función del DAO. Sin embargo, el recurso de obtener la lista de propuestas en base a unos criterios también accede al método del DAO de obtener todas las propuestas en el caso de que no se haya especificado ningún criterio con el que aplicar el correspondiente filtro, como ya se ha mencionado en el modelo de CU de BilbozkatuBB.

A continuación se muestra este diagrama reducido en la Figura 45. Sin embargo, en el **Anexo A** adjunto al presente documento se muestra el diagrama de clases extendido en el que se incluye la relación de las clases ejecutoras del servicio (*Applicaion, WebbApplicationInitializer* y *PostgresCloudConfig* si se ejecuta en la nube) con las clases del *framework* Spring y de Spring Boot que toman parte.

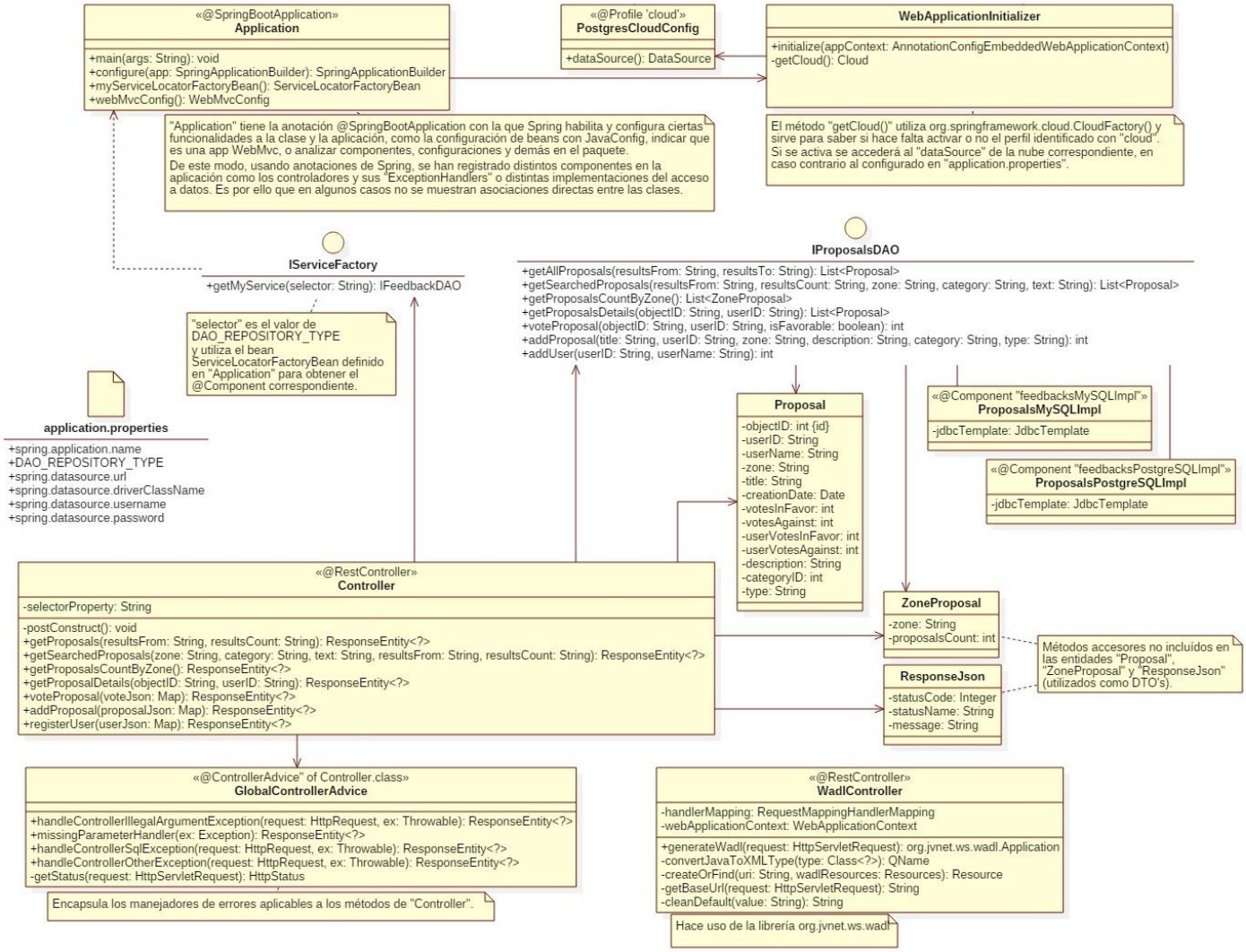


Figura 45: Diagrama de Clases reducido de BilbozkatuBB

Las explicaciones en relación a la composición de las capas software, la configuración del *DataSource* en local o en la nube, así como la configuración de dicho origen de datos como base de datos MySQL o PostgreSQL, entre otros, se ha analizado en profundidad en el apartado 6.1.2.3 del diagrama de clases de *UsersFeedbackBB*, y son aplicables al de la Figura 45.

6.2.2.4. Diagrama de secuencia

El flujo de eventos o la secuencia seguida en cada uno de los accesos a los recursos del servicio es muy similar, por no decir igual, entre ellos y con los de *UsersFeedbackBB*. A modo de ejemplo, a pesar de ser similar al mostrado en la Figura 26, seguidamente se muestra un diagrama de secuencia simplificado de BilbozkatuBB para el recurso de obtener todas las propuestas, con los datos generales de las mismas para su previsualización en la pantalla correspondiente de la app Bilbozkatu, como se verá en el siguiente apartado 6.3.

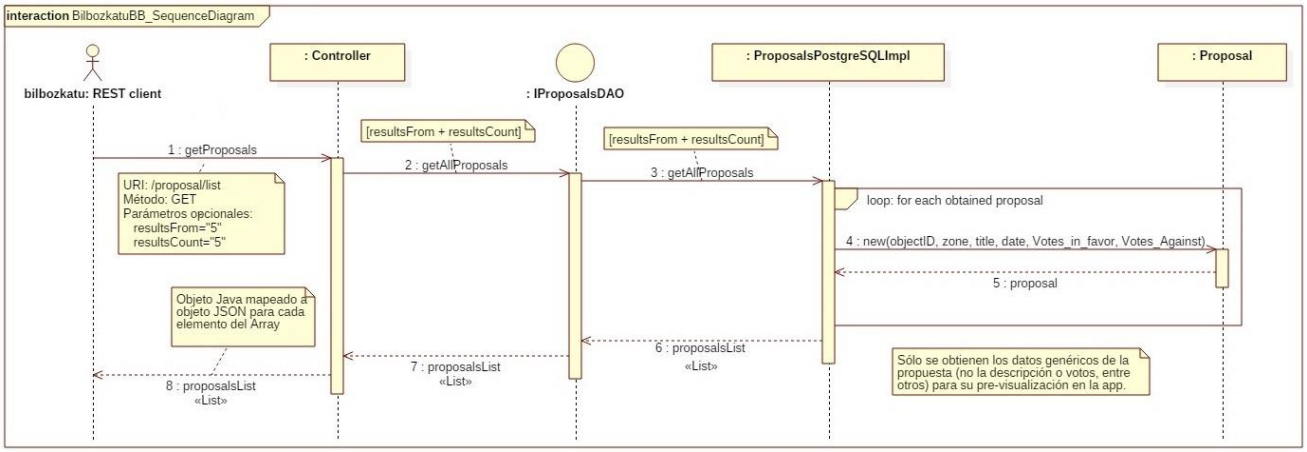


Figura 46: Diagrama de secuencia simplificado de BilbozkatuBB

Como se puede comprobar en la Figura 46, el flujo de eventos presenta muchas similitudes con el del diagrama de la Figura 26 antes comentado. En este caso, por comentar algo que no había sido mostrado anteriormente, la capa de acceso a datos (la cual en el diagrama se muestra como la interfaz ProposalsDAO y su implementación ProposalsPostgreSQLImpl, aunque entre estas dos no hay interacción porque realmente son lo mismo a nivel de Java por así decirlo) crea una lista (mediante la interfaz List<T> de Java) de propuestas. Esta lista se devuelve a través de las distintas capas software hasta que el controlador la *mapea* a un array de objetos JSON correspondientes a cada propuesta obtenida en base a los parámetros opcionales *resultsFrom* y *resultsCount.*

6.2.2.5. Modelo de datos

La realización del modelo de datos requiere de algunas subfases: análisis de requerimientos, diseño conceptual, selección del SGBD, diseño lógico (en el que se normaliza y se definen las restricciones oportunas) y finalmente el diseño físico. A continuación se describen estos pasos brevemente pero analizando todos los campos a tener en cuenta.

Si bien el diseño del modelo de datos es algo más complejo que el de *UsersFeedbackBB*, tampoco tiene demasiada dificultad. En cuanto al **análisis de requerimientos** estos son los que se han ido comentando en las secciones anteriores relativas al análisis y diseño del servicio. Se deben gestionar las propuestas que los ciudadanos (usuarios) hacen indicando algunos campos como el *título*, la *descripción*, la *fecha de creación*, la *zona*, *categoría* o *tipo*. En este caso la *zona* y *categoría* son datos gestionados y configurables a través de la app Bilbozkatu, de modo que el servicio sólo debe guardar el dato recibido sin tener que restringirlo. Por otro lado, el *tipo* actualmente sólo puede ser "ciudadano", pero se ha definido para dejar abierta la posibilidad de que el Ayto. de Bilbao pueda crear propuestas también en el futuro.

Además del usuario que ha creado la propuesta también debe ser posible gestionar los votos que los usuarios hacen de las mismas, a favor o en contra. Sólo se permite un voto por usuario y persona, y también se almacena la fecha de realización del voto.

Una vez descritos los requerimientos del modelo a diseñar, el siguiente paso consiste en la realización del **diseño conceptual**, en el que se desarrolla el modelo de Entidad-Relación que se puede apreciar a continuación en la Figura 47.

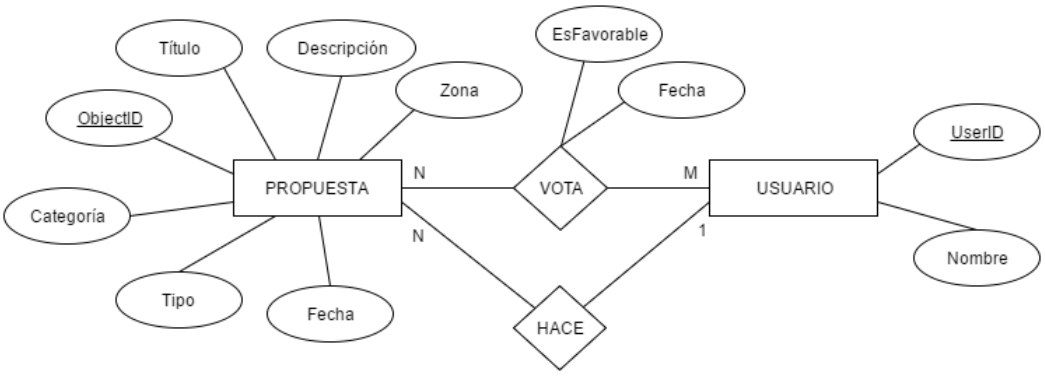


Figura 47: Modelo Entidad-Relación de BilbozkatuBB

El Modelo Entidad-Relación no presenta complejidad alguna. Consta de dos entidades, Propuesta y Usuario, las cuales está relacionadas por la relación, valga la redundancia, llamada *Vota* y por *Hace*. Así, un usuario puede crear y votar las propuestas que quiera y una propuesta, que será siempre de un usuario, puede ser votada por más de uno pero tan sólo puede existir una relación de voto entre un usuario y una propuesta, en la que se almacena el sentido del voto (a favor o en contra) y la fecha de realización del mismo.

Por otro lado los datos a guardar por parte de las entidades son los indicados en el modelo de la Figura 47: un título, descripción, zona, categoría, tipo (sólo "ciudadano" de momento) y fecha para las propuestas, y nombre para los usuarios, además de los correspondientes campos clave con los identificadores en cada una de ellas.

A continuación se procedería al **diseño lógico** del modelo de datos (modelo relacional), incluyendo después la normalización, vistas o restricciones si las hubiera. Del Modelo de Entidad-Relación anterior se extrae lo siguiente:

PROPUESTA(<u>ObjectID</u>, UserID, Título, Descripción, Zona, Categoría, Tipo, Fecha)
CE: USU. (de la relación "Hace")

USUARIO(<u>UserID</u>, Nombre)

VOTO(<u>ObjectID</u>, <u>UserID</u>, EsFavorable, Fecha) (de la relación "Vota")
CE: PROP. CE: USU.

El diseño lógico que se ha hecho, junto con la simplicidad del diseño conceptual por no haber por ejemplo atributos multi-evaluados, hacen que en este punto ya se cumplan las **formas normales**. No se definen **vistas** ya que no se ha considerado necesario. No hay consultas muy complejas que hacer contra la base de datos, ni hace falta proteger partes de la base de datos o dividirla para su uso por parte de distintos tipos de usuarios de la base de datos. Estas razones, entre otras, hacen que directamente se vaya a la siguiente fase: la elección del Sistema de Gestión de Bases de Datos.

En cuanto a la **elección del SGBD** se aplica lo mismo que ya se ha comentado para el modelo de datos de *UsersFeedbackBB* del apartado 6.1.2.5. En resumen, si bien inicialmente se comenzó con MySQL, la necesidad de despliegue del servicio en Cloud Foundry hizo que el servicio también soportara PostgreSQL, ambas bases de datos relacionales como se ha ido comentando. Esto lleva al diseño físico de la base de datos, tanto para MySQL como para PostgreSQL., tal y como se puede ver en los siguientes esquemas.

MySQL

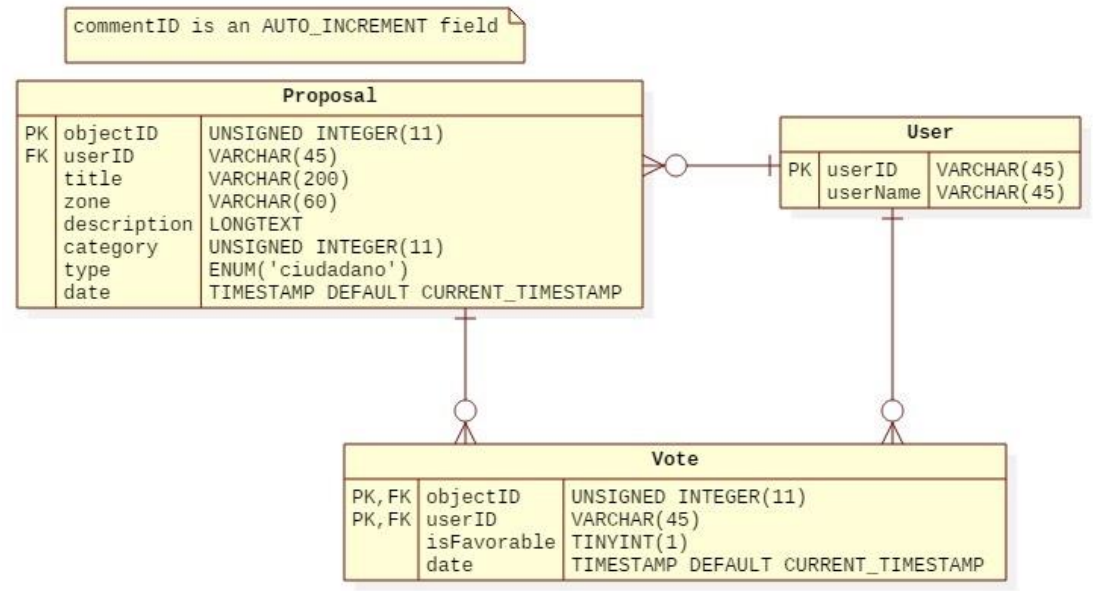


Figura 48: Diseño físico en MySQL para BilbozkatuBB

En estos diseños físicos de las Figuras 48 y 49 se puede apreciar el diseño lógico antes realizado. También se aprecian los tipos de los campos definidos, específicos para cada tipo de base de datos pero equivalentes entre sí gracias a las restricciones definidas en PostgreSQL.

En la siguiente Figura 49 se muestran las restricciones definidas en los campos *category* y *type* con el objetivo de que los campos sean equivalentes con respecto a los tipos de datos definidos para MySQL. Además, se han creado índices para *title* y *description* para mejorar el rendimiento al comparar con dichos campos en las consultas que lo hagan.

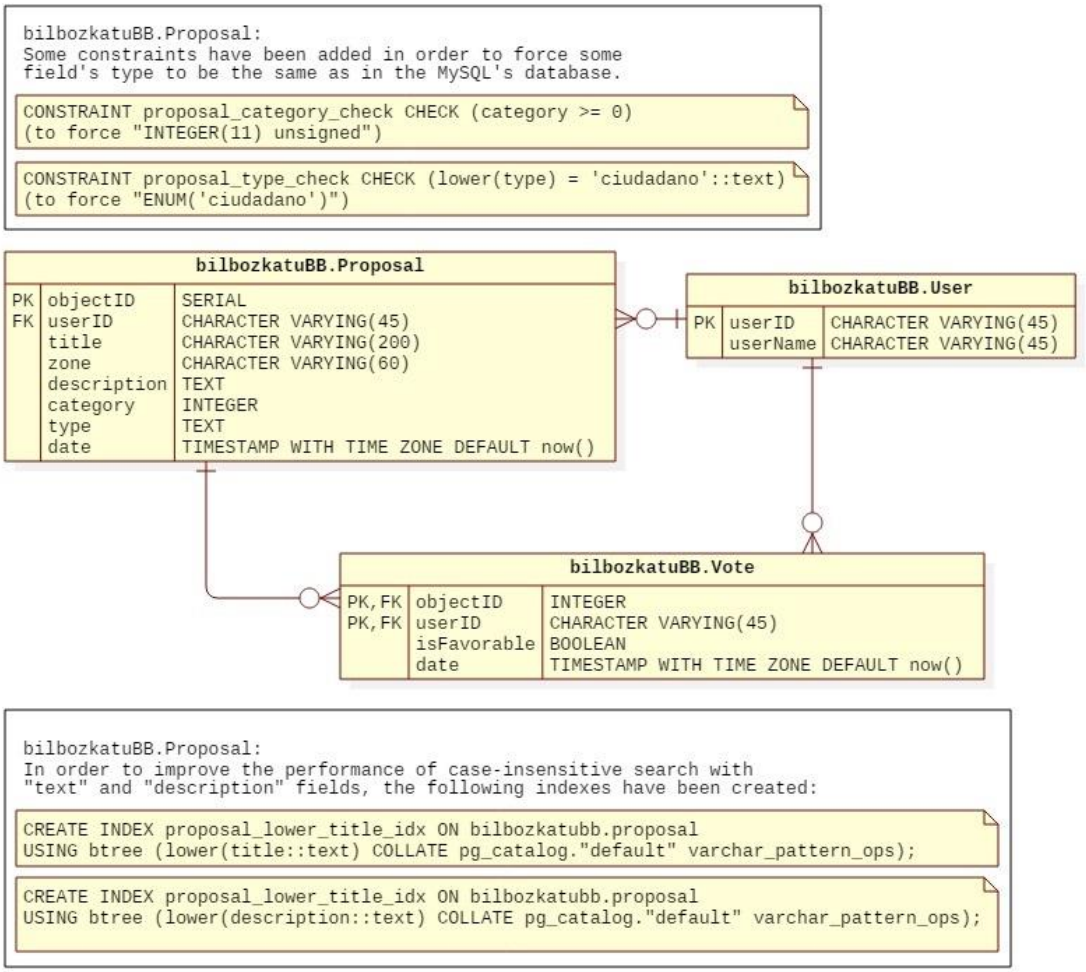


Figura 49: Diseño físico en PostgreSQL para BilbozkatuBB

Se adjuntan en el **Anexo B** los *scripts* SQL completos utilizados para la creación del modelo de datos de *BilbozkatuBB*, tanto para MySQL como para PostgreSQL.

6.2.3. **Implementación**

La implementación se ha hecho del mismo modo que en *UsersFeedbackBB*, en una serie de fases incrementales con soluciones parciales en cada una de ellas. Es aplicable todo lo comentado en el apartado 6.1.3, como la utilización del *framework* Spring, Spring Boot, Maven o la plataforma Cloud Foundry para el despliegue del servicio. En esta sección se menciona algún aspecto específico de *BilbozkatuBB*, pero referenciando lo ya comentado para *UsersFeedbackBB.*

6.2.3.1. Arquitectura

Toda la arquitectura utilizada en este servicio está recogida en la sección 6.1.3, como se ha comentado. Se ha hecho uso del *framework* Spring, y en concreto del nuevo componente llamado Spring Boot con el que facilitar el proceso de creación de aplicaciones y servicios Spring. Este servicio, además, se ha desarrollado siguiendo el estilo de arquitectura software llamado RESTful.

La estructura de ficheros es exactamente la misma que la empleada en *UsersFeedbackBB*, con ligeras modificaciones en el nombre de algunas clases o variables. Es por ello que no se vuelven a analizar cuáles son y para qué sirven los paquetes y clases del servicio, tan sólo se muestra dicha estructura en la siguiente Figura 50.

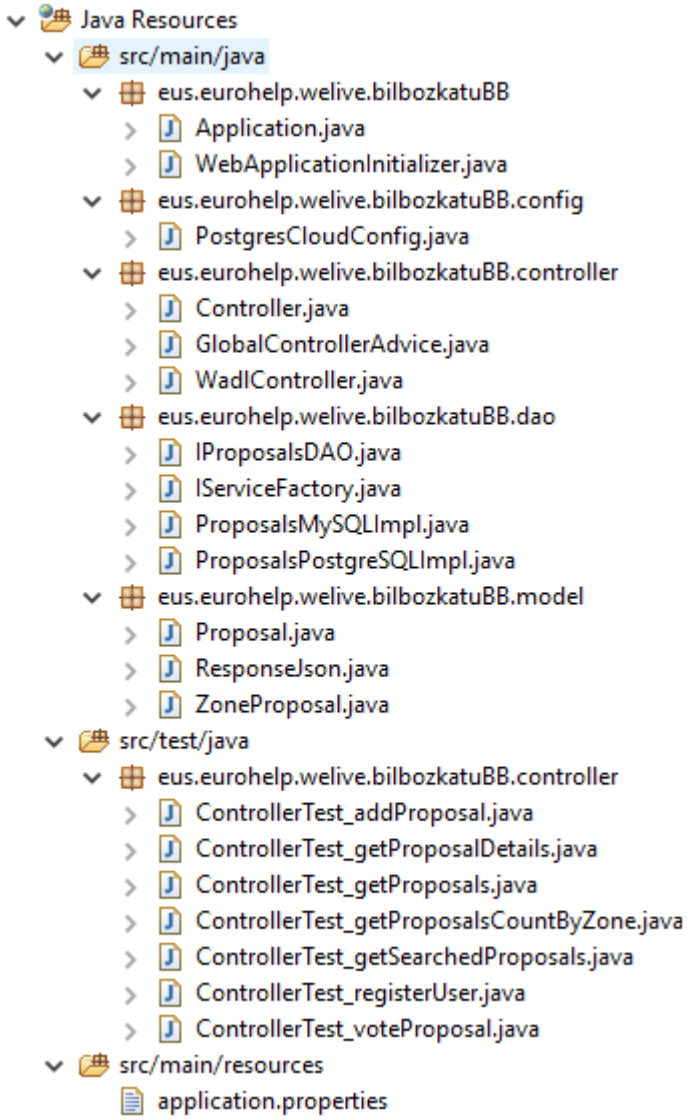


Figura 50: Estructura de ficheros de BilbozkatuBB

Técnicas empleadas

Este apartado de técnicas empleadas tampoco incluye nada específico de este servicio. Se ha desarrollado y utilizado de nuevo el **Patrón Factoría** de Spring ya explicado, haciéndolo del mismo modo. No obstante, cabe añadir por ejemplo el uso de la paginación, también llamado *cursoring*, mediante los parámetros *resultsFrom* y *resultsCount* mencionados en el análisis y en el apartado de identificación de recursos. Gracias a estos parámetros opcionales desde la app se pueden obtener las propuestas por bloques, de más nuevos a más viejos, para

poder implementar en la misma la funcionalidad de *infinite-scroll* de manera que el usuario para bajando en la lista mientras se van cargando dinámicamente más propuestas viejas. En este caso la paginación se ha realizado indicando el *id* de una propuesta como filtro para indicar a partir de cuál se desean obtener nuevos comentarios. En *UsersFeedbackBB*, sin embargo, esto se gestiona directamente con las fechas (en vez de indicar el *id* de un comentario, se puede indicar la fecha a partir de la cual o desde la cual obtener más *feedbacks*).

6.2.3.2. Problemas encontrados

Los problemas encontrados son los mismos que en el punto 6.1.3 excepto el tratamiento de fechas, ya que como se acaba de explicar la paginación se ha realizado con los propios *id's* de las propuestas. La formación en el uso de Spring, Spring Boot, técnicas como el Patrón Factoría de Spring, peculiaridades de PostgreSQL frente a MySQL, aprendizaje de la instalación y uso de PostgreSQL o la realización de alguna sentencia SQL algo más compleja han sido algunas de las dificultades que se han tenido en el desarrollo de este servicio.

6.2.4. **Verificación y pruebas**

De nuevo la herramienta **JUnit** ha sido la utilizada para la realización de pruebas unitarias de todos los recursos ofrecidos por el servicio, tanto para la validación de los parámetros de las llamadas a la REST API, como para la verificación de las propias funcionalidades. Dichas pruebas, agrupadas por ficheros correspondientes a cada recurso, están en la misma ubicación en la estructura de ficheros que en *UsersFeedbackBB*. A continuación se muestran en la Figura 51 los resultados de la ejecución de todas las pruebas, ejecutables por otro lado de manera independiente y sin alterar en ningún momento el estado de la base de datos (gracias a la ejecución automática de *rollback* sobre la misma al terminar la ejecución de las pruebas unitarias).

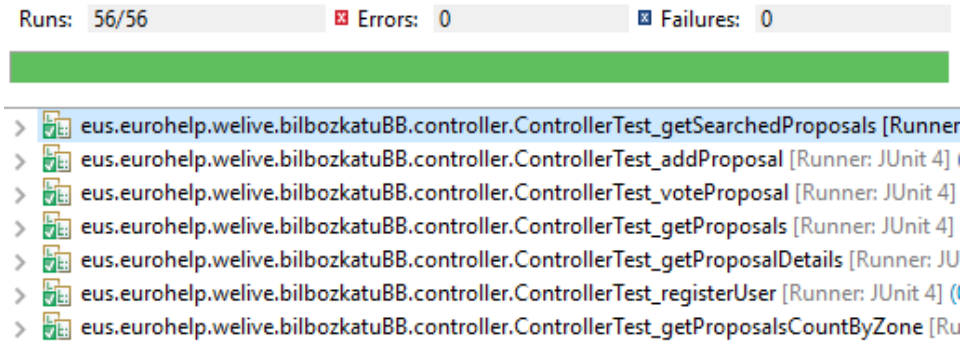


Figura 51: Resultados JUnit de BilbozkatuBB

Cabe añadir que los propios testeos que se han realizado de las apps indirectamente también verifican el correcto funcionamiento de este servicio web.

6.2.5. **Despliegue (Cloud Foundry)**

Para terminar con los *Building Blocks* desarrollados en general y con *BilbozkatuBB* en concreto, el despliegue de este servicio se ha realizado de nuevo en la plataforma Cloud Foundry. Sirve para ello todo lo comentado en el apartado 6.1.5 en relación a dicha plataforma y los pasos a seguir para el despliegue. De este modo el servicio se ha dejado disponible en la nube para su uso por parte de las apps de WeLive, Bilbozkatu en este caso. El servicio se ha dejado disponible bajo el dominio ***https://{nombre-servicio}.cloudfoundry.welive.eu***.

6.3. App Bilbozkatu

Una vez analizados los artefactos software correspondientes a los *Building* Blocks desarrollados, en este punto 6.3 y en el 6.4 se procede a analizar las apps desarrolladas: Bilbozkatu y BilbOn.

Figura 52: Logo App Bilbozkatu

Para ello se analizarán las fases de análisis, diseño, implementación, verificación y publicación, mostrando en ellas aspectos como una descripción general de la aplicación, los diseños de las pantallas realizadas, los componentes de la plataforma de WeLive u otras APIs que utiliza, o los pasos a seguir para la publicación de la app en Google Play. Como ya se ha comentado en el alcance del proyecto, por el momento sólo está disponible en Android).

Es importante destacar que en el punto 6.3.2.2 donde se analizan las pantallas se muestran las funcionalidades a modo de **manual del usuario**, a fin de facilitar el uso de esta aplicación móvil.

6.3.1. **Análisis**

En esta sección se detalla la especificación y requisitos de la app Bilbozkatu, tanto desde el punto de vista de los componentes externos a usar (de la plataforma WeLive o no), como desde el de las propias funcionalidades que la app tiene.

6.3.1.1. Especificación y requisitos

A grandes rasgos, Bilbozkatu es una aplicación móvil híbrida a través de la cual los usuarios de Bilbao pueden sugerir propuestas sobre cualquier barrio de la ciudad, catalogándolas en alguna categoría entre "Medio ambiente", "Movilidad", "Infraestructura", "Servicios sociales", "Deportes", "Presupuestos" u "Otros. Los

demás usuarios tienen la opción de ver las propuestas realizadas por otros usuarios e incluso filtrarlas. También permite a los usuarios enviar comentarios sobre una propuesta (compuestos por un comentario y un *rating* del 1 al 5) o votar una propuesta a favor o en contra, de manera que el Ayto. de Bilbao tenga la opinión de los ciudadanos y actúe, si lo cree oportuno, en consecuencia.

Requisitos técnicos

Como se ha ido comentando en los primeros capítulos del presente documento, son requisito de la empresa Eurohelp las siguientes tecnologías y herramientas para el desarrollo de las apps de WeLive:

- La app debe ser desarrollada utilizando **Ionic** como *framework front-end*, cosa que indirectamente requiere de hacer uso de **Apache Cordova** para realizar una implementación híbrida de la app utilizando tecnologías web para ello. Además, para poder aprovechar al máximo las opciones de Ionic, el *framework* de desarrollo de la propia app es **AngularJS**, siendo el MVC (Modelo-Vista-Controlador) o MVVC el modelo empleado.
- El despliegue de la app se debe hacer en la tienda de Android Google Play, para lo cual la app se debe compilar para dicha plataforma a través de Apache Cordova.

Requisitos funcionales

En líneas generales las funcionalidades o características que la app debe tener son las siguientes:

- Inicialmente se muestra un ***splash screen*** con los efectos, logotipos y textos correspondientes a WeLive, el Ayto. de Bilbao y la Comisión Europea.
- La primera vez se muestran los **Términos y Condiciones** de uso de la aplicación, que deben ser aceptados para acceder a la app.
- En la aplicación hay disponible un **menú lateral** desplegable desde el cual poder acceder a distintas pantallas (inicio de sesión, búsqueda de propuestas, formulario de nueva propuesta, mapa de búsqueda de propuestas y la pantalla *About*), además de poder cambiar de idioma.
- Tiene soporte **multilenguaje**: castellano, euskera e inglés.
- Dispone de una pantalla inicial en la que poder **buscar y visualizar las propuestas** creadas por los ciudadanos en forma de lista (indicando sólo algunos datos generales de previsualización de las mismas). Se pueden **ordenar** en función de algunos parámetros (más nuevas primero, por título o zona en orden alfabético, mejor valoradas en cuando al *rating* de los *feedbacks* que tenga, por más votos a favor y en contra). También es posible **filtrarlas** en función de la **zona, categoría, texto o una combinación** de ellos.

- El **filtro por zona** también se puede hacer de forma más visual en la pantalla del **mapa**, en el que aparecen marcadas las zonas o barrios de los que poder enviar propuestas de mejora, y por cada uno se muestra el número de propuestas que tiene. Si el usuario pulsa el marcador de una zona se abre la pantalla del punto anterior, con la lista de propuestas filtradas ya por esa zona.
- El usuario puede seleccionar una propuesta de la lista para ver sus detalles y acceder a nuevas funcionalidades:
    - Se muestra el **estado** de la propuesta, abierta o cerrada, si la propuesta ya ha caducado. Es decir, se calcula si ya ha expirado en función de la fecha de creación, fecha actual, y el periodo de validez configurado en la app (2 meses inicialmente).
    - Si el usuario está autenticado y la propuesta en estado "abierta" (no ha expirado), el usuario puede **enviar su voto** a favor o en contra, sólo una vez. En todo momento verá el sentido de su voto.
    - Si el usuario está autenticado y la propuesta "abierta", el usuario puede **enviar los *feedbacks* que quiera**, compuesto cada uno por un texto y un *rating* (del 1 al 5) sobre la propuesta.
    - El usuario puede **ver los *feedbacks*** enviados por otros usuarios, y la **media del *rating*** de los mismos. También puede **ver el nº de votos a favor y en contra** que tiene.
    - El usuario puede acceder a una segunda pantalla de detalles, en la que se muestran **otras estadísticas**. También se visualizan de manera gráfica el nº de votos.
- Si el usuario está autenticado puede enviar una nueva propuesta, indicando para ello el título, descripción, zona y categoría (estos dos últimos eligiendo de una serie de posibles opciones).
- El usuario puede autenticarse a través de un botón en el menú: se hace uso del **componente AAC (*Authentication and Authorization Control System BB*) de WeLive** para la autenticación de usuarios.
- Dispone de una pantalla ***About*** en la que se muestra **información de la aplicación** y desde la que el usuario puede **ejecutar un cuestionario** sobre la app hasta un máximo de cuatro veces.
- Se hace uso del **componente *Logging BB* de WeLive** para registrar ciertos eventos en relación a la actividad de la aplicación, por ejemplo, "app iniciada" o "usuario x autenticado". Para ello la aplicación debe autenticarse también ante la plataforma WeLive (distinto a la autenticación del usuario, pero a través del mismo componente AAC BB).

Componentes externos

Dada la naturaleza de la aplicación se hace uso de diferentes componentes, APIs y librerías externas, las cuales han requerido un estudio importante en cuanto a su especificación, uso e integración. Además, en relación a las librerías JavaScript tanto para Ionic como para AngularJS, también se ha realizado una tarea de investigación para el descubrimiento de las mismas. Así, los componentes externos utilizados son:

- Plataforma WeLive
  - **AAC BB** para la autenticación de usuarios.
  - **LoggingBB** como servicio para registrar una serie de *logs* o KPIs definidos para la app.
  - Componente para mostrar el **cuestionario** de la app (para registrar algunos datos por parte de los usuarios y conocer su opinión acerca de la utilidad y usabilidad de la app).
  - El ***UsersFeedbackBB*** desarrollado, aunque no está integrado en el *marketplace* de WeLive.
  - El ***BilbozkatuBB*** desarrollado, aunque no está integrado en el *marketaplace* de WeLive.
- Librerías JavaScript a través del gestor Bower
  - **angular-local-storage** para el acceso al *localStorage* o datos almacenados de la app en el dispositivo.
  - **angular-translate** y **angular-translate-loader-static-files** para implementar el multilenguaje en la app, teniendo para ello un fichero de traducciones por cada idioma integrado.
  - **angular-chart** (que requiere chart.js) para una visualización gráfica animada de algunos datos, en este caso los resultados de una votación.
  - **angular-momentjs** (que requiere moment.js) para un cálculo correcto del periodo de validez de una propuesta (y verificar que una fecha está o no dentro del intervalo de los últimos dos meses con respecto de la fecha actual).
  - **ionic-ratings** para visualizar un componente con el que indicar un *rating* del 1 al 5 mediante iconos de estrellas.
  - **ionic-modal-select** para mostrar un panel predefinido en el que se muestran los idiomas disponibles en la app y fácilmente configurable.

    - **ionic-platform-web-client** para utilizar el servicio Ionic View de la plataforma Ionic.io y poder compartir y visualizar fácilmente la app en un dispositivo móvil a modo de prueba.
- Otras librerías JavaScript
    - **Google Maps JavaScript API v3**

El uso e implementación, incluyendo diagramas de secuencia, de algunos de los componentes más importantes mencionados se analizan en el apartado correspondiente a la implementación en el punto 6.3.3. En esa sección también se detallarán los plugins de Apache Cordova instalados y usados en la app, de manera que el desarrollo híbrido mediante tecnologías web sea complementado por esos plugins que hacen uso de características nativas de la plataforma para la que se va a compilar la app, Android en este caso.

6.3.2. **Diseño**

En este punto se muestra el diseño de la app, comenzando por el modelo de CU y terminando por el diseño de la interfaz gráfica de cada pantalla desarrollada. Debido a la complejidad y numerosos detalles que están implementados en la app (navegación dependiendo de algunas condiciones, control de errores, detalles de mejora de interacción con el usuario, etc.), en este apartado no se muestran diagramas de clases o de secuencia tal y como se ha hecho para los *Building Blocks* implementados.

No obstante, en el siguiente punto correspondiente a la implementación se repasa la arquitectura y estructura de ficheros, sus responsabilidades y objetivos. También se incluyen algunos diagramas de secuencia para algunos aspectos importantes o destacables en el flujo de algunos procesos.

6.3.2.1. Modelo de Casos de Uso

En el modelo de CU que se muestra en la siguiente Figura 53 se pueden apreciar algunos de los CU definidos en los *Building Blocks*, si bien es cierto que en la app de algunos de ellos se ha extraído algún que otro CU nuevo destacable e importante.

Por ejemplo, los CU "Ver propuestas" y "Ver propuestas por criterios" en la app provocan que el usuario también pueda "Ver votos de propuesta". Esto sucede en la pantalla en la que se muestra la lista (filtrada o no) de propuestas que se verá más adelante, en la que por cada una se previsualiza no sólo el título o la zona como ya se ha comentado anteriormente, sino también el nº de votos que la propuesta tiene tanto a favor como en contra. Sucede exactamente lo mismo con "Ver media de ratings" (se debe recordar que un *feedback* en *UsersFeedbackBB* está compuesto por el comentario en sí, y un *rating* del 1 al 5). De modo que una propuesta es valorada por los ciudadanos tanto en *rating*

como en votos, aunque lo importante de cara al Ayto. de Bilbao son las votaciones.

Por otro lado la necesidad de identificarse para poder crear una propuesta, votar o crear *feedbacks* se expresa mediante las anotaciones como <<*extend*>>, tal y como se anota en el propio diagrama.

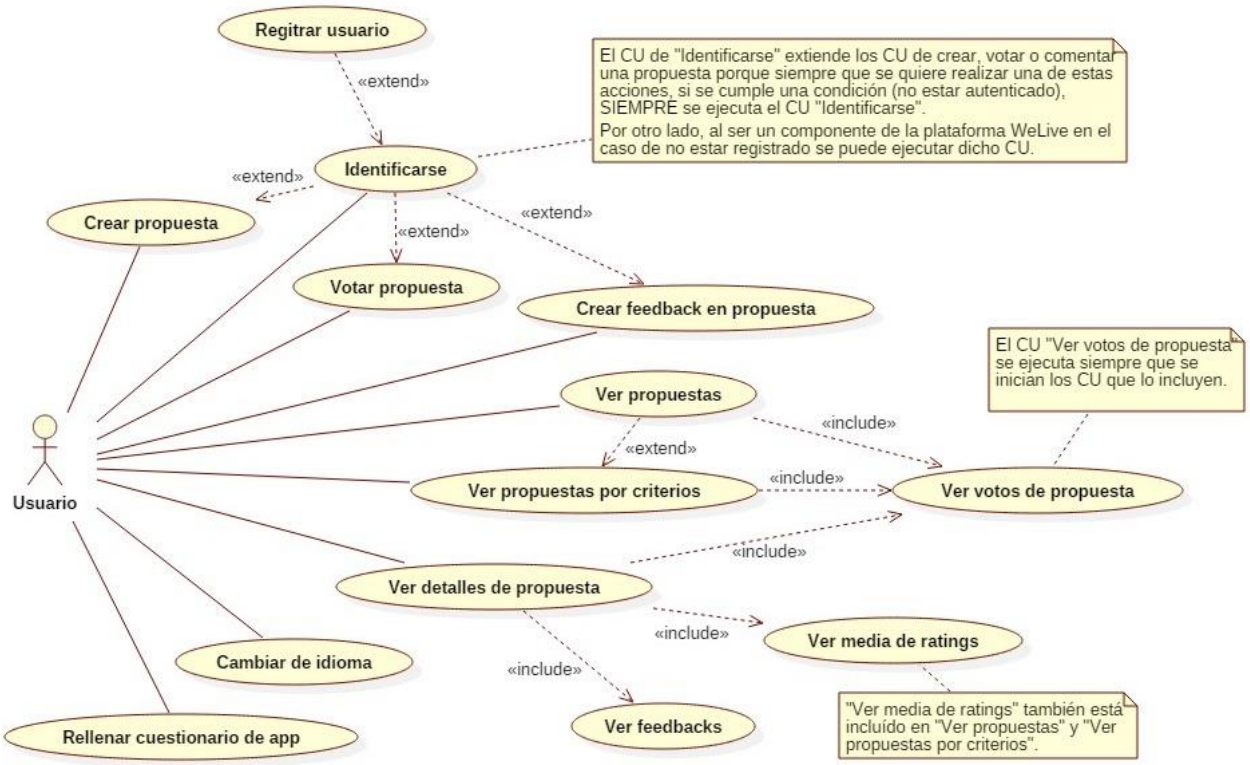


Figura 53: Modelo de Casos de Uso de la app Bilbozkatu

6.3.2.2. Diseño de la interfaz gráfica

En este segundo punto de la fase de diseño de la app se muestran los diseños de las interfaces gráficas desarrolladas junto con algunos comentarios explicativos de los CU. Es importante destacar que el diseño (en cuanto a componentes y estilos utilizados) de algunas pantallas fue inicialmente realizado por **Alex Novoa**, diseñador gráfico de Eurohelp de la sede de Bilbao. Esto es así por comenzar con el desarrollo de la primera pantalla de la app sobre una base, ya que el autor del presente proyecto ha tenido que formarse en numerosas herramientas y tecnologías y evitar así empezar completamente de cero. Las interfaces (y estilos) desarrolladas inicialmente por Alex Novoa son:

- Splash screen (mostrado en el arranque de la app).
- Lista de propuestas

No obstante, dichos diseños han sido cambiados a lo largo del desarrollo y el resto de pantallas sí se han generado desde cero, prevaleciendo siempre los

tonos de colores sugeridos por Alex en concordancia con el Ayto. de Bilbao, cuyo color principal es el rojo.

Cabe añadir que, si bien en el presente documento sólo se pueden mostrar las interfaces una vez cargadas, existen diferentes tipos de transiciones entre pantallas que hacen más atractiva la visualización de la app para el usuario, gracias a Ionic.

Splash screen

Figura 54: Pantalla splash screen de Bilbozkatu

Tal y como se ha explicado, las dos pantallas pertenecientes al *splash screen* han sido realizadas por el diseñador Alex Novoa de Eurohelp. Al arrancar la aplicación se muestra la pantalla de la izquierda de la Figura 54 que es igual en todas las apps de WeLive, y pasados unos instantes se muestra la segunda pantalla con transiciones de por medio, mostrando el logotipo y título de la aplicación.

Términos y Condiciones de uso

Una vez terminado el *splash screen*, se muestra una pantalla en la que el usuario pueda leer los términos y condiciones de uso de la aplicación. Dichos términos

han sido redactados por una comisión especializada dentro del proyecto WeLive, de acuerdo a la legislación pertinente en relación, por ejemplo, a la LOPD (Ley Orgánica de Protección de Datos).

El usuario debe aceptar dichos términos y condiciones para poder acceder a la app. Si lo hace, dicha aceptación queda almacenada en el dispositivo y no volverá a aparecer en siguientes ejecuciones de la app. En caso contrario se muestra un mensaje informativo y se cierra la aplicación.

Figura 55: Pantalla de Términos de Uso de Bilbozkatu

Lista de propuestas

La pantalla de lista de propuestas es la principal en la app Bilbozkatu, y la que se muestra al arrancar la app después del *splash screen* y, si fuera necesario, la pantalla de términos y condiciones. En las Figuras 56, 57 y 58 se aprecia que en la parte superior están las opciones para filtrar las propuestas por zona, categoría y texto, y en la inferior la propia lista de propuestas si las hay, o un mensaje informativo en caso contrario.

En la imagen de la derecha de la Figura 56, en la parte superior derecha se aprecia un botón con el que poder abrir un pequeño panel desde el cual el usuario puede reordenar las propuestas en función de diversos criterios.

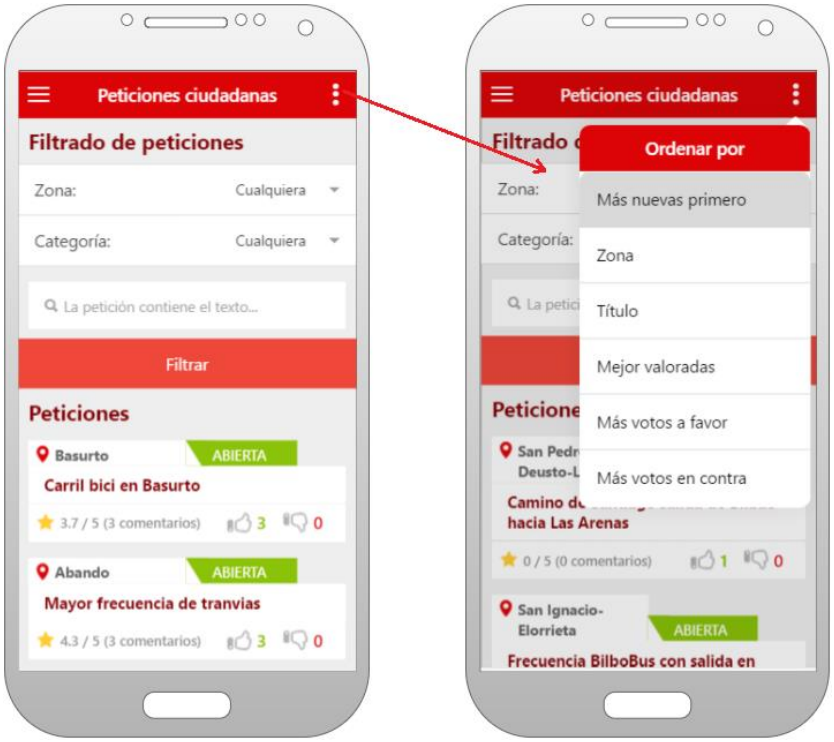


Figura 56: Pantalla de lista de propuestas (1/4)

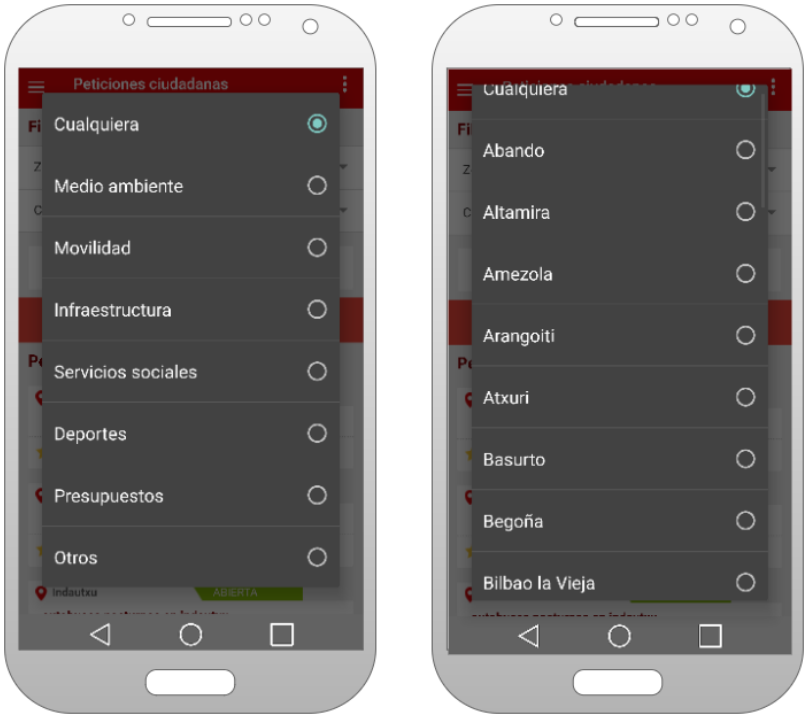


Figura 57: Pantalla de lista de propuestas (2/4)

En la Figura 57 se muestra el control de selección nativo de Android, si bien la implementación se hizo con componentes de Ionic con tecnologías web. Esta es otra muestra del desarrollo híbrido de las apps.

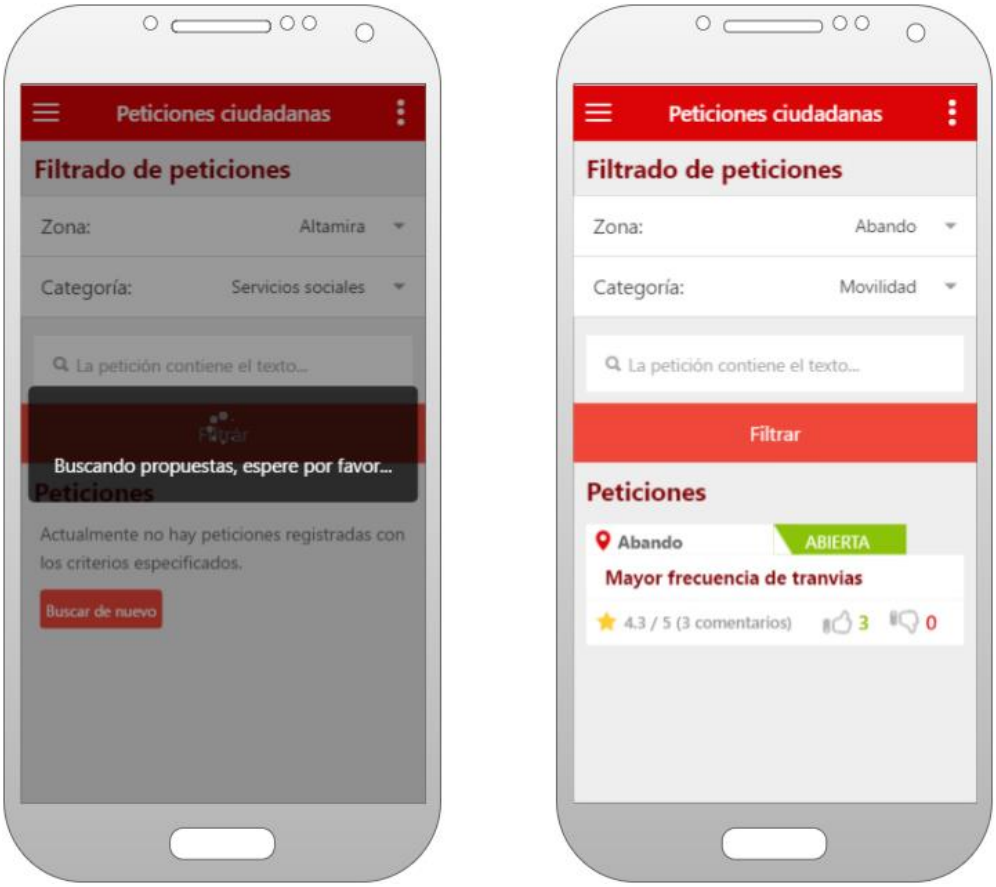


Figura 58: Pantalla de lista de propuestas (3/4)

En la Figura 58 aparece cómo se visualiza la recarga de propuestas y el mensaje si no las hay, y también la propuesta encontrada filtrando por el barrio "Abando" y la categoría "Movilidad". En todas las interfaces se puede apreciar la constante en las apps de WeLive para Bilbao, el uso de tonos rojos para menús, mensajes, botones, etc.

Figura 59: Pantalla de lista de propuestas (4/4)

Por último se muestra una última Figura, la 59, en la que se aprecia en la parte inferior el elemento de carga implementado con Ionic. Mientras el usuario baja en la lista nuevos bloques de propuestas viejas son buscados a modo de paginación.

También se incluye un botón para subir al inicio de la lista de manera sencilla para el usuario, y no tener que recorrer de nuevo la lista hacia arriba con de manera manual.

Menú

En la parte izquierda de la barra superior de todas las pantallas se muestra un icono con el que abrir el menú lateral izquierdo, excepto en la de Detalles y Estadísticas, en las cuales se muestra la opción para ir "Atrás". En la siguiente Figura 60 se muestra dicho icono marcado con un cuadrado azul.

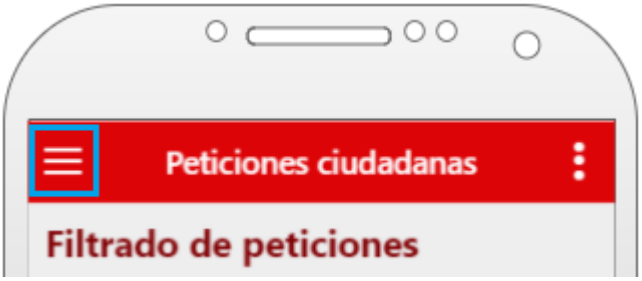


Figura 60: Barra superior de Bilbozkatu

Y, a continuación, se muestra el menú lateral. Desde él se puede acceder a las pantallas de inicio de sesión, lista de peticiones o propuestas analizada en el punto anterior, formulario de nueva petición, el mapa, la pantalla *About* o de información y términos de uso, y el panel de selección del idioma.

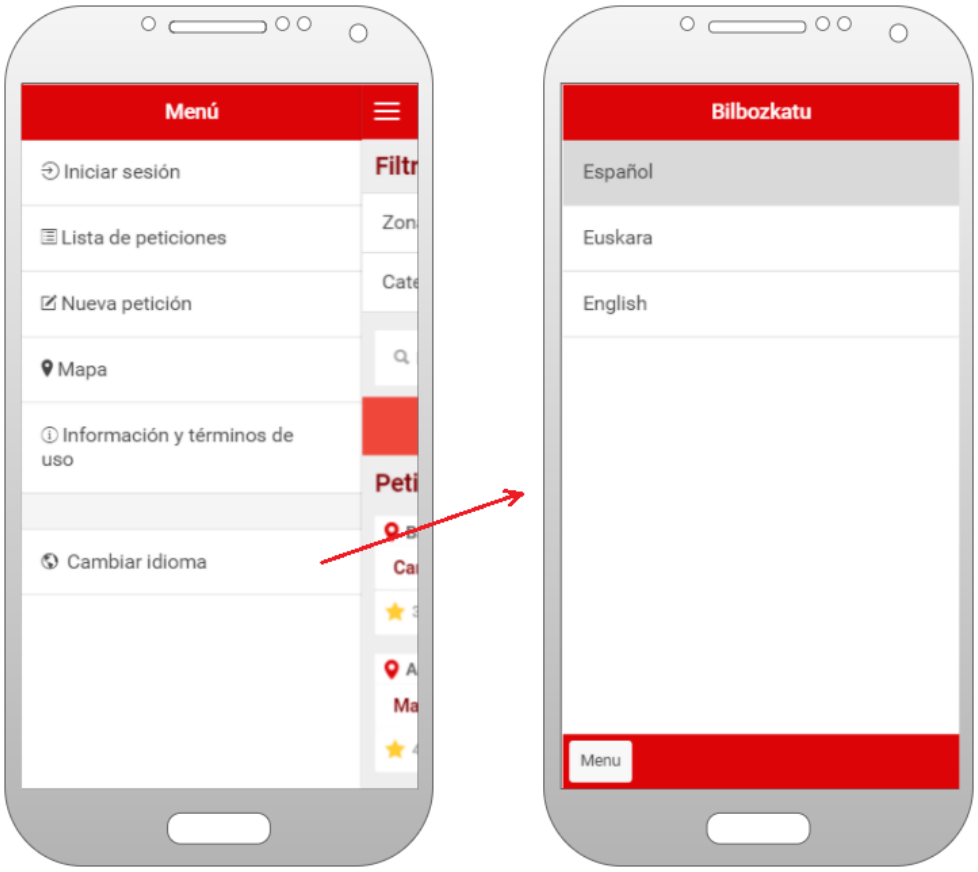


Figura 61: Interfaz del menú e idiomas de Bilbozkatu (1/2)

En el caso de que el usuario esté autenticado, la opción de "Iniciar sesión" se sustituye automáticamente por "Cerrar sesión", botón que realiza dicha acción y haría que se mostrara de nuevo la opción "Iniciar sesión". Esto se demuestra en la siguiente Figura 62, marcado por un cuadrado azul.

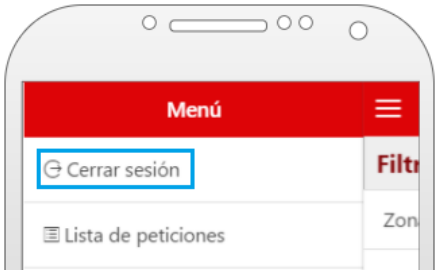


Figura 62: Interfaz del menú de Bilbozkatu (2/2)

Pantalla del mapa

En esta pantalla se muestra un mapa centrado en la ciudad de Bilbao, y sobre él se muestran los marcadores de las zonas o barrios predefinidos en la app. Al pulsar en un marcador se muestra el nombre de dicha zona y el número de peticiones registradas en la misma. En caso de que haya al menos una, además de mostrar el icono del marcador en rojo, se puede pulsar sobre el enlace del nº de peticiones (ver pantalla izquierda de la Figura 63) para acceder directamente a la pantalla de lista de propuestas ya descrita, filtrando automáticamente las peticiones por esa zona.

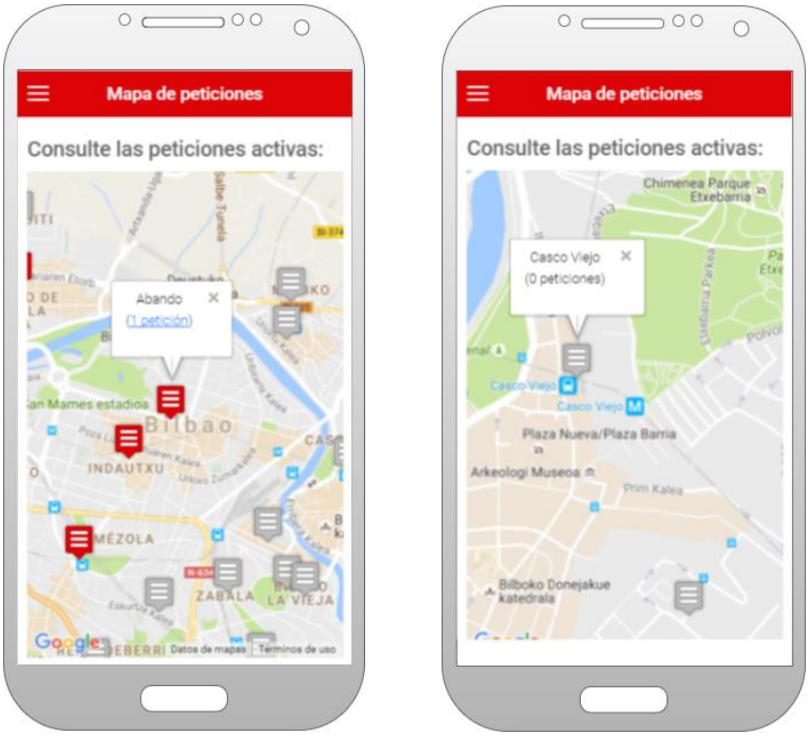


Figura 63: Pantalla de mapa de propuestas de Bilbozkatu

Detalles de una propuesta

Pulsando sobre una propuesta en la pantalla de lista de peticiones se accede a la pantalla de detalles de una propuesta. En ella se pueden ver la mayoría de los datos almacenados de la misma: el título, la descripción, los resultados provisionales de los votos, el estado (abierta o cerrada en función de si ha expirado o no), una imagen predefinida asociada a la categoría a la que pertenece y los datos relativos a los *feedbacks* de la petición. Sobre estos últimos debajo del título, a la izquierda de los votos, se muestra la media de *rating* calculado con los comentarios, y en la parte inferior se muestran los propios comentarios (texto y *rating* en cada uno, además del nombre del usuario que lo ha hecho y la fecha y hora de creación.

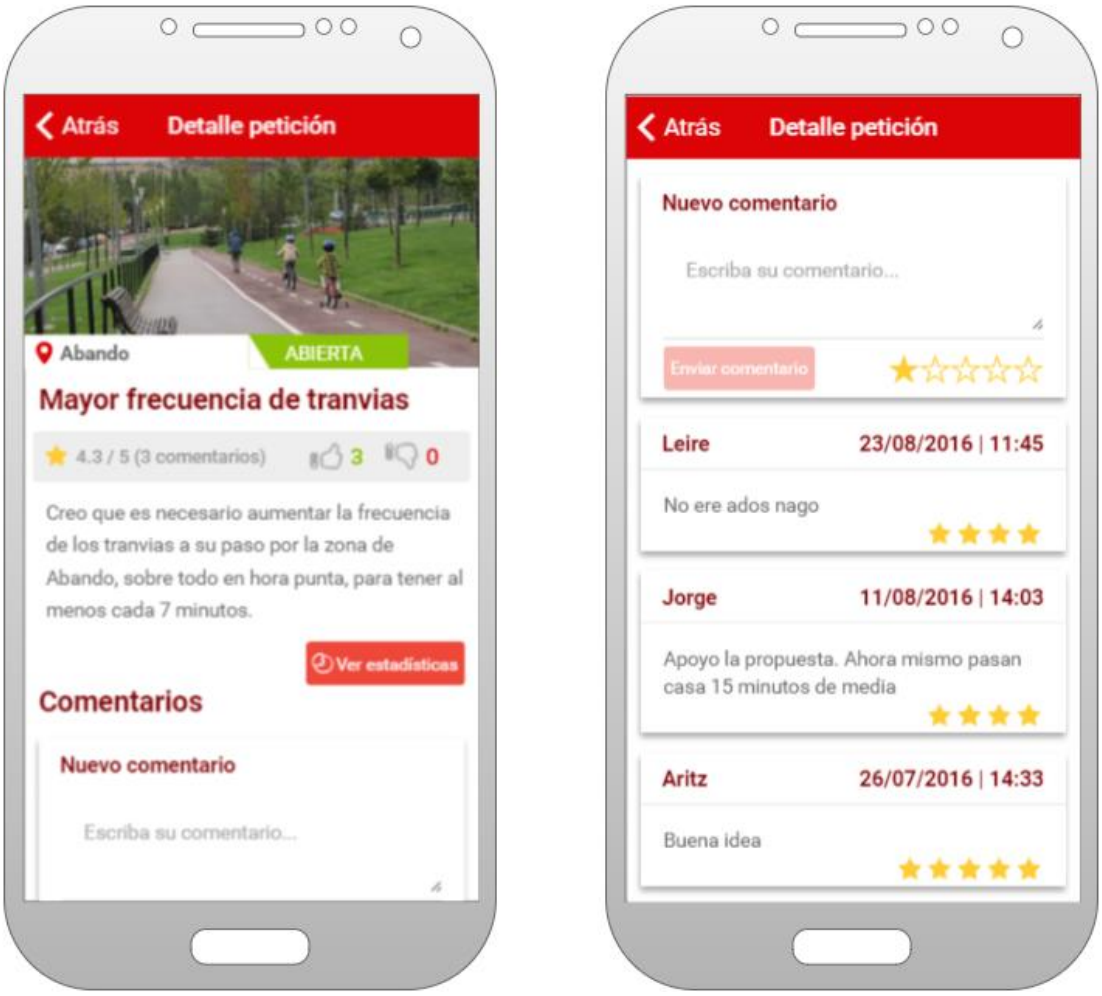


Figura 64: Pantalla de detalles de una propuesta de Bilbozkatu

Al igual que con la lista de propuestas, la lista de *feedbacks* está implementada también con la técnica llamada ***cursoring*** para la paginación. Inicialmente se obtiene un bloque con los comentarios más nuevos, y el usuario según alcanza el último de ellos se obtiene un nuevo bloque de comentarios más viejos hasta llegar al primero que se envió. Por otro lado, debido a que es factible que se generen nuevos comentarios una vez el usuario ha cargado la pantalla de

detalles de una propuesta y, con ello, la lista de *feedbacks*, periódicamente se comprueba la existencia de nuevos comentarios. Si existen nuevos comentarios en la parte superior de la lista se mostraría un botón para cargarlos, y en caso de pulsarlo se insertarían en la parte superior manteniendo la ordenación cronológica de los mismos.

Finalmente, cabe añadir que se muestra un botón para acceder a la pantalla de estadísticas (ver Figura 64) que se analiza en el siguiente punto. Además, en esta pantalla el usuario puede crear un *feedback* o votar la propuesta, utilizando el pequeño formulario al inicio de la sección de comentarios y pulsando los iconos de voto a favor o en contra de la parte superior respectivamente. No obstante, estas dos funcionalidades requieren que el usuario esté autenticado, de modo que en caso de necesitarlo la app muestra un mensaje como el de la Figura 65, desde el cual poder acceder a la pantalla de inicio de sesión y, una vez finalizado el proceso, redirigir a la propuesta.

Figura 65: Mensaje de autenticación en la pantalla de detalles de Bilbozkatu

Estadísticas de una propuesta

A través del botón "Ver estadísticas" de la pantalla de detalles (ver Figura 64) se acceden a más detalles y estadísticas, y a algunas ya mostradas pero con otro formato. Así, tal y como se puede ver en la Figura 66, se muestra gráficamente el resultado de los votos, y los datos detallados de nº de votos a favor, en contra, media de *rating* de los *feedbacks*, el tipo, categoría, estado (abierta o no), el nombre del usuario que la creó y la fecha de creación.

Figura 66: Pantalla de estadísticas de Bilbozkatu

Figura 67: Gráfica de la pantalla de detalles de Bilbozkatu

La carga de la gráfica se realiza de manera animada con un efecto en la transiciónque la rellena progresivamente. Además, pulsando sobre la sección verde o roja se muestran también el nº de votos a favor y en contra que los ciudadanos han enviado, respectivamente. En caso de no tener votos, se muestra una gráfica por defecto en color gris.

Pantalla de inicio de sesión

El inicio de sesión en la plataforma de WeLive se ha realizado a través del componente AAC BB de WeLive. Los detalles del flujo desarrollado se analizarán en el siguiente apartado 6.3.3 de implementación, pero en relación al diseño de la interfaz no se ha tenido que realizar nada en este caso. A continuación se muestra la pantalla en la que sólo se ha añadido un botón con el que ejecutar el proceso que se realiza en un *inAppBrowser*. Es decir, al ejecutar el login en Bilbozkatu se accede al componente AAC de WeLive a través de un navegador incrustado en la propia app, de manera que todo el contenido mostrado en dicho navegador corresponde al componente de WeLive y no a la app.

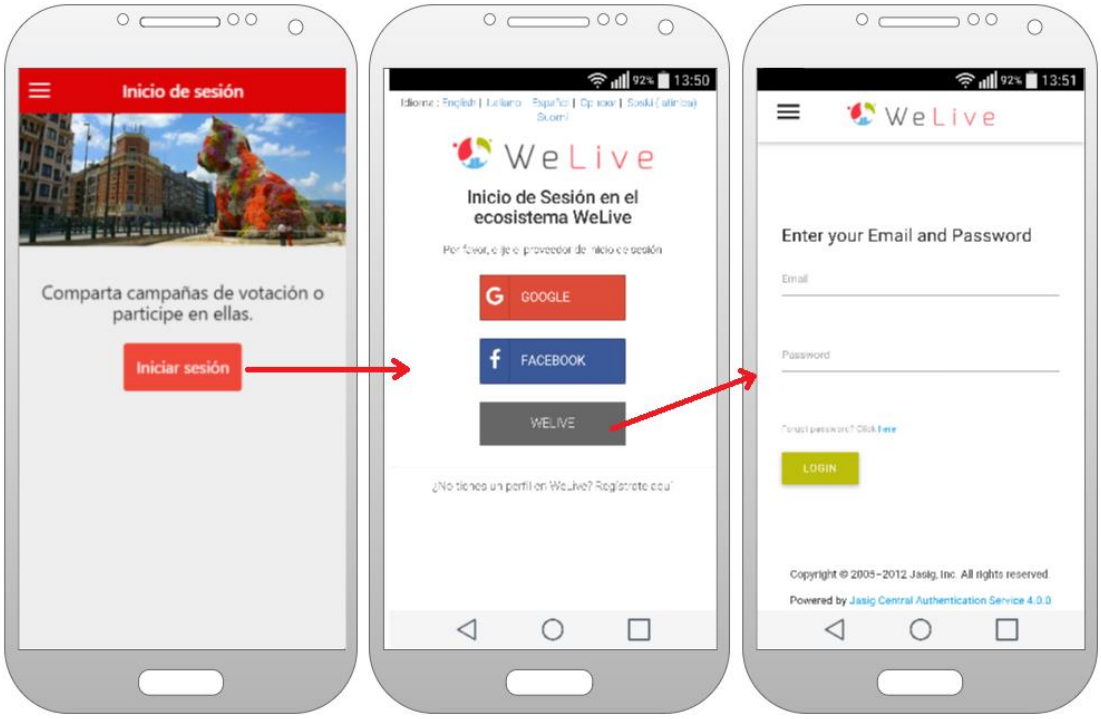


Figura 68: Pantallas de proceso de autenticación (1/2)

Por tanto, en la anterior Figura 68 se muestran las pantallas por las que pasar en el proceso de autenticación en la plataforma WeLive. Comenzando desde la app, el botón "Iniciar sesión" ejecuta el navegador interno de la app y a partir de ahí se muestra el componente AAC de WeLive. Primero se debe seleccionar un proveedor de identidad (por ejemplo, WeLive) y a continuación se muestra el formulario en el que introducir las credenciales.

Una vez hecho eso, se muestra una última pantalla en la que el usuario debe aceptar los permisos requeridos en la app. Estos permisos, tal y como se ha explicado en el apartado 5.2.1, se han tenido que configurar previamente. En este caso la autorización es la de leer el perfil básico de un usuario. Por último, el control vuelve a la app Bilbozkatu y se muestra durante unos instantes el mensaje satisfactorio de inicio de sesión (ver Figura 69).

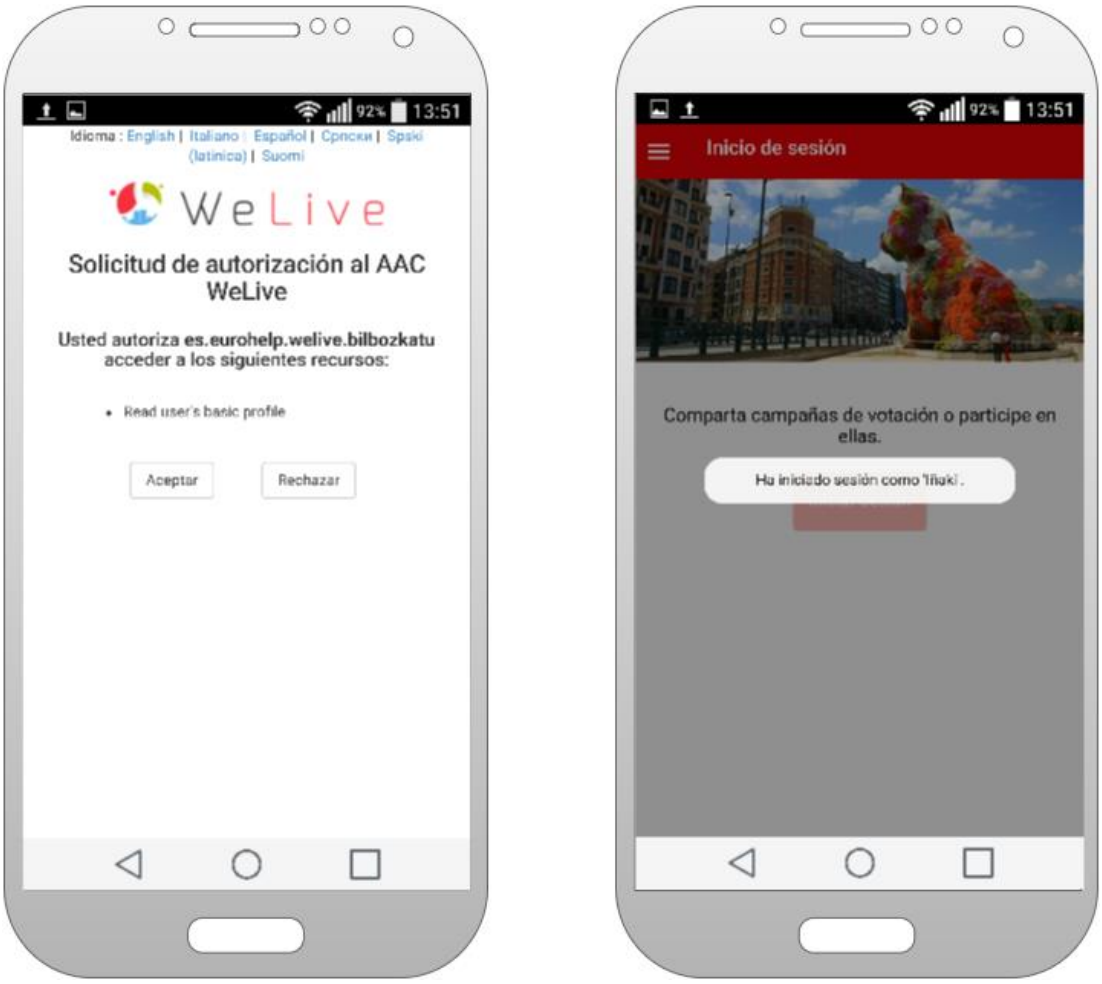


Figura 69: Pantallas de proceso de autenticación (2/2)

En este punto, una vez terminado el proceso de login de WeLive pero antes de mostrar al usuario el mensaje de autenticación satisfactoria, la app internamente ha registrado el identificador del usuario generado en el *Building Block BilbozkatuBB*, si no existe ya, a fin de controlar qué usuarios han creado propuestas o las han votado, como se ha comentado en apartados anteriores.

Pantalla de creación de una propuesta

En esta pantalla se muestra un formulario para la creación de nuevas propuestas o peticiones por parte de los ciudadanos. En él se debe indicar el título y descripción de la propuesta a crear, y la zona y la categoría a la que pertenece. Estos dos últimos parámetros tienen una serie de opciones ya que están predefinidos en la app, aunque pueden modificarse sin problema en nuevas versiones de la misma.

En la imagen izquierda de la siguiente Figura 70 se puede apreciar que el botón de envío queda inhabilitado hasta que todos los campos hayan sido definidos, ya que son obligatorios. El usuario debe estar autenticado, mostrándose en ese caso un mensaje como el de la Figura 65 que permita ejecutar el login.

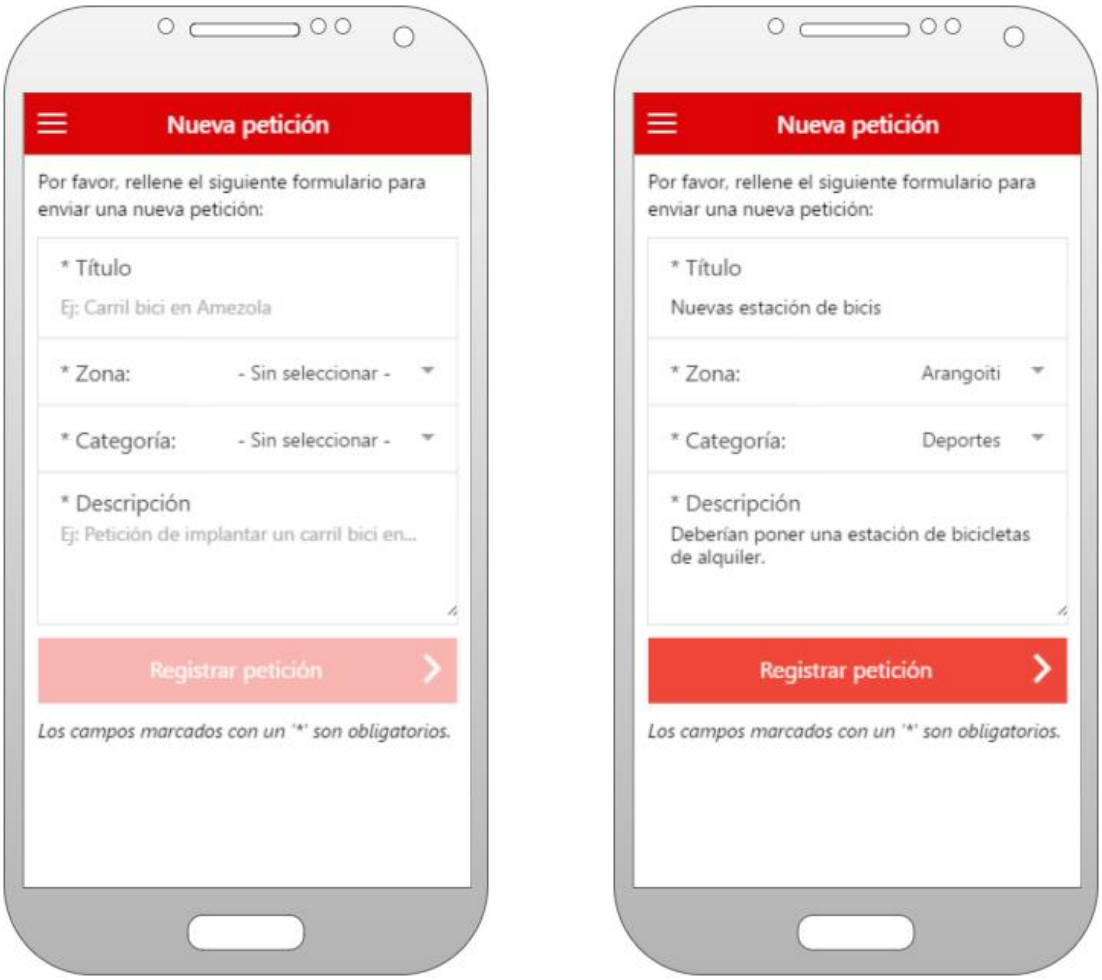


Figura 70: Pantalla de creación de propuesta (1/2)

En los campos del formulario se realizan las validaciones necesarias. Si algún campo está mal completado se indica mediante un mensaje informativo en rojo debajo del dato inválido, como se muestra en la siguiente Figura 71.

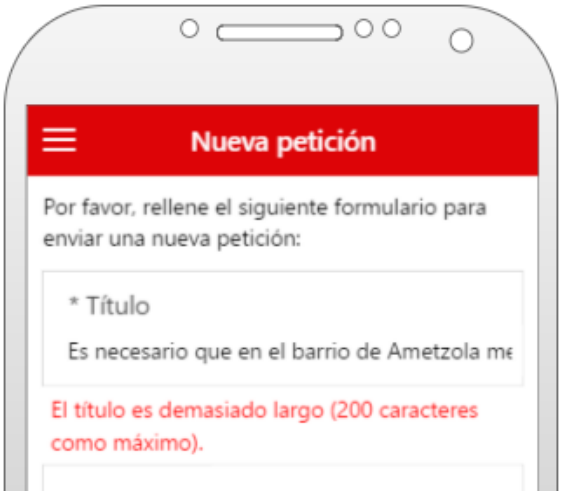


Figura 71: Pantalla de creación de propuesta (2/2)

Pantalla de Información y Términos de uso

Por último, en la pantalla *About* o de información y términos de uso se muestra una imagen decorativa de Bilbao, una sección en la que ejecutar un cuestionario de opinión acerca de la app, y el texto y logotipos correspondientes (ver la Figura 72).

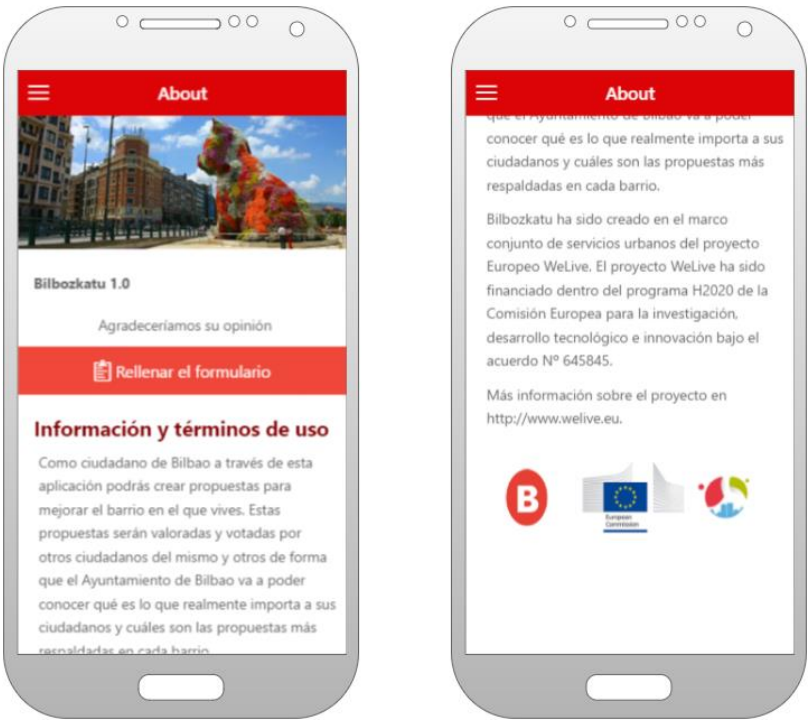
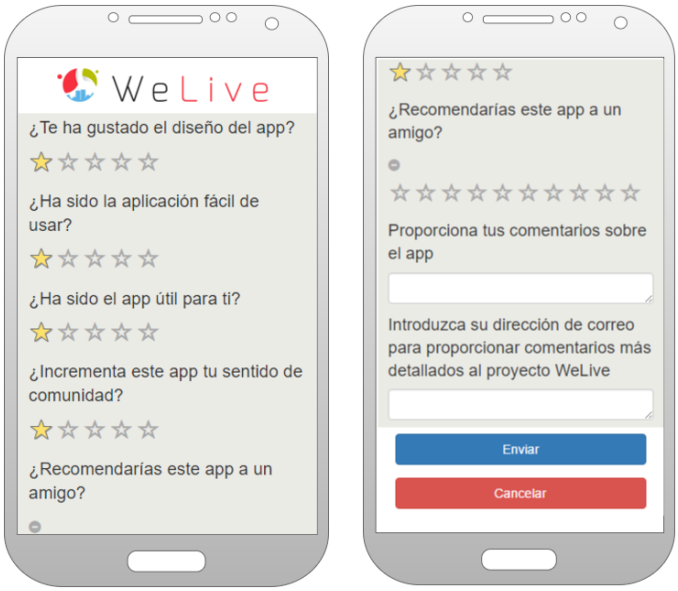


Figura 72: Pantalla de About de BilbozkatuBB



El cuestionario se ejecuta del mismo modo que el AAC BB de WeLive, en un navegador interno de la app. Su realización y, por tanto, diseño de la interfaz, etc., ha sido realizada por terceras personas también. En la app tan sólo se ejecuta y se gestiona la respuesta para mostrar al usuario si se ha realizado correctamente o no.

Figura 73: Interfaz del cuestionario de opinión de Bilbozkatu

6.3.3. **Implementación**

Una vez repasadas las interfaces gráficas de la app, así como el modo de uso de las mismas y aspectos más importantes de la navegación, en este apartado se comentan algunos puntos importantes en relación a la implementación de la app. Cabe destacar que el desarrollo de Bilbozkatu ha supuesto un gran esfuerzo en el aprendizaje de numerosas tecnologías, herramientas, *frameworks*, conceptos y técnicas nuevas para el autor del presente proyecto, y en ocasiones novedosas en el mundo del software en general, por no hablar del estudio e integración de componentes externos en colaboración en algunos casos con otras empresas. Es por ello que a continuación sólo se menciona algo más detalladamente la implementación de algún que otro punto relevante.

Se mencionará la arquitectura general de la app, la estructura de ficheros y los *plugins* de Apache Cordova utilizados. Seguidamente se analizarán las partes del código referentes al proceso de inicio de sesión mediante OAuth haciendo uso del componente de WeLive AAC BB, incluyendo un diagrama de flujo de eventos, además de otros extractos de código interesantes. Finalmente se incluyen algunos comentarios sobre técnicas utilizadas.

Desarrollo, arquitectura y estructura

El desarrollo de la app se ha llevado a cabo de igual manera a los anteriores artefactos software analizados, es decir, en una serie de fases incrementales en las que se han ido obteniendo versiones de la app cada vez con más funcionalidad.

En cuanto a la arquitectura, todo lo relacionado al respecto se puede ver en el **capítulo 5** del presente documento, en concreto en la **sección 5.1** relativa al *framework* de Ionic. En dicha sección se analizan de manera detallada todos los aspectos en cuanto a la arquitectura de una app con Ionic y de otros *frameworks* de los cuales Ionic es una capa superior, como son Apache Cordova (y uso de sus *plugins*) y AngularJS para dotar del paradigma MVC (Modelo-Vista-Controlador) o MVVC al desarrollo. La arquitectura de dichos *frameworks* y la integración entre ellos, pues, ya se ha explicado y a continuación se muestra la estructura de ficheros de la app junto con los objetivos y responsabilidades que se han definido y que separan en capas el código implementado.

Los distintos componentes se han definido e implementado en la medida de lo posible de acuerdo a los patrones y buenas prácticas definidas para AngularJS, con lo que se consigue un código mucho más legible y ordenado. Algunas de estas buenas prácticas se han ido incorporando a lo largo del desarrollo según se han ido estudiando, debido a que son muchos los conceptos que se han tenido que estudiar desde el principio y no es posible aplicarlo todo al inicio de la fase de implementación por desconocimiento. Sin embargo, en siguientes proyectos en la empresa ya en desarrollo ya se emplea este conocimiento.

Dicho esto, la estructura general del proyecto es la mostrada en la siguiente Figura 74, junto con descripciones generales de cada parte del mismo debido a la cantidad de ficheros existentes:

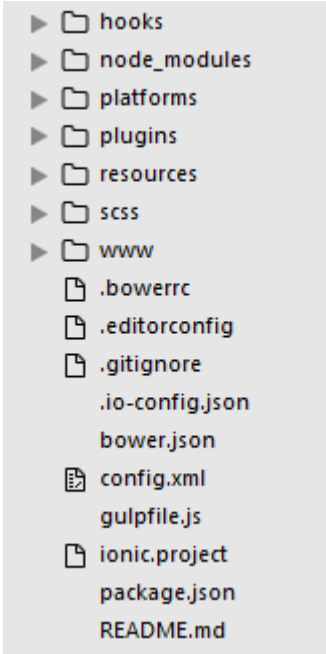


Figura 74: Estructura general de la app Bilbozkatu en Ionic

En la estructura general de una aplicación híbrida hecha con Ionic se encuentran los directorios y ficheros de la imagen anterior. Al iniciar la implementación se ha utilizado el CLI de Ionic para genera una aplicación con una estructura de ficheros básica que ha estado en constante evolución. Los directorios *hooks/* y *node_modules/* no han sufrido variación alguna (son usados por Cordova e Ionic para ejecutar con comandos específicos), al igual que *scss/* (lenguaje css que no se ha necesitado en la app). Otros directorios y ficheros son los siguientes, en los cuales se puede intuir que también es un proyecto Cordova para el desarrollo de una app híbrida con tecnologías web (no app web para móvil):

- ***platforms/***: directorio en el que se guarda el código y compila la app para las plataformas indicadas, en este caso Android. En él se obtiene el fichero .apk de Android para la instalación de la misma.
- ***plugins/***: directorio donde se almacenan los *plugins* de Cordova e Ionic instalados en la app, para utilizar características específicas de plataforma, y que se acceden al compilarla.
- ***resources/***: directorio en el que poder guardar recursos (imágenes, etc.) de cara a utilizarlos en la compilación de la app, como el icono.
- ***www/***: directorio que contiene el código de la app (JavaScript, librerías, CSS, imágenes, etc.).

- ***bower.json***: fichero donde se definen las dependencias Bower (gestor de librerías JavaScript).
- ***config.xml***: fichero de configuración de Cordova (se pueden indicar parámetros para la compilación de la app, específicos o no de una plataforma en concreto, como la versión de la app, el nombre, autor, algunos permisos de la app, configuraciones como permitir sólo la visualización *portrait* (vertical) en el dispositivo, plugins de Cordova a utilizar o el icono de la app).
- ***ionic.project***: configuración de Ionic (en este caso la configuración para gestionar la app con Ionic View asociado a una cuenta de Ionic.io.
- ***package.json***: dependencias de Node.js.

Ahora, en cuanto a la estructura de la propia aplicación (*www/*), se encuentran los siguientes bloques en la Figura 75:

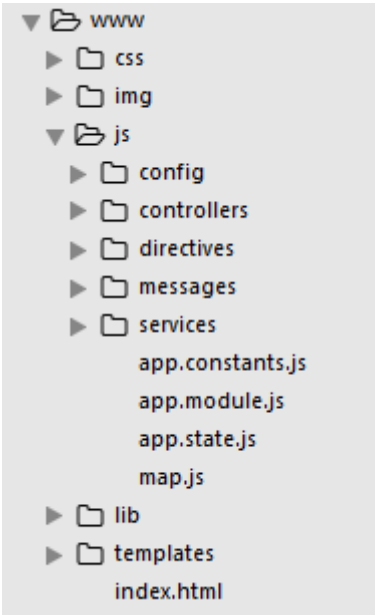


Figura 75: Estructura de la app Bilbozkatu

Se muestran los directorios css/, img/ y templates/ para las hojas de estilo, imágenes y plantillas HTML de la app respectivamente. En este caso le estructura importante es la definida en el directorio js/ que incluye todos los ficheros JavaScript implementados (en *lib/* se encuentran las librerías instaladas para su uso en AngularJS, Ionic o como complemento, enumeradas ya en el apartado 6.3.1.1 del análisis de Bilbozkatu).

Es importante señalar que, junto a lo mencionado en relación a la arquitectura de aplicaciones AngularJS en el capítulo 5, se trata de una aplicación denominada ***Single Page*** ya que toda la aplicación se integra en un mismo HTML. Este HTML es el *index.html* y en él, además de referenciar todos los

ficheros (css, js) que la app necesita incluir, sólo dispone de una etiqueta *<body/>* en la que se inyectan las demás plantillas dinámicamente. Esto sucede gracias al ***AngularJS Routing*** y ***Templating*** de Angular, pero en este segundo caso además se hace también a través de **Ionic** como se puede ver en la siguiente imagen.

```
<body ng-app="bilbozkatuApp">

  <!-- Comment below to disable splash screen -->
  <div id="splash" class="animated fadeOutUp">
     <div id="splash_logo" class="animated bounceInDown"></div>
     <div id="splash_text" class="animated_fast fadeInUp"></div>
     <div id="splash_sub_text" class="animated_fast fadeInUp"></div>
  </div>
  <div id="legals" class="animated_fast fadeOutLeft"></div>

  <ion-nav-view></ion-nav-view>
</body>
```

Figura 76: Extracto del index.html de Bilbozkatu

La etiqueta ***ion-nav-view*** de Ionic permite insertar en ella las demás plantillas HTML. Pero, antes de hacerlo, es necesario explicar que existe una plantilla **abstracta**, que se inyecta en el *index.html* y permanece constantemente, de manera que las demás plantillas se añadan a ella y se muestre una composición de ambas. La plantilla abstracta corresponde a la barra superior y menú de la app, y el resto de ellas se inyectan de manera dinámica (aunque es posible por HTML o JavaScript modificar esa plantilla "padre" desde una normal.

```
$stateProvider

    // setup an abstract state for showing the menu
    .state('app', {
        url: '/app',
        abstract: true,
        templateUrl: 'templates/menu.html',
        controller: 'AppCtrl'
    })

    .state('app.terms', {
        url: '/terms',
        views: {
        'menuContent': {
                templateUrl: 'templates/terms.html',
                controller: 'TermsCtrl'
            }
        },
        resolve: TermsController.resolve
    })

    .state('app.proposalslist', {
        url: '/proposals?zone',
        //cache: false,
        views: {
            'menuContent': {
                templateUrl: 'templates/proposals.html',
                controller: 'ProposalsListCtrl'
            }
        }
    })
```

Figura 77: Extracto de uso de AngularUI Router en Bilbozkatu

Antes de analizar la estructura de ficheros JavaScript se debe mencionar también que para conseguir esto se deben definir estados utilizando *AngularJS UI Router* (ver la Figura 77). En dicho extracto se muestran algunos de los *estados* definidos para la app. En concreto:

- ***"app"***: estado abstracto para definir la plantilla "padre" bajo la cual se insertan las demás plantillas. Se asocia con *menu.html* (ya que la barra superior como el menú lateral se muestra siempre) y el controlador *AppCtrl* definido.

```
<ion-side-menus enable-menu-with-back-views="false">

  <ion-side-menu-content>

    <ion-nav-bar class="bar-stable">
      <!-- Botón "Atrás" a mostrar cuando corresponda -->
      <ion-nav-back-button>
        {{ 'all-menu.back-button-label' | translate }}
      </ion-nav-back-button>
      <!-- Botón para abrir el menú lateral -->
      <ion-nav-buttons side="left">
        <button class="button button-icon button-clear ion-navicon"
          menu-toggle="left"></button>
      </ion-nav-buttons>
    </ion-nav-bar>

    <ion-nav-view name="menuContent"></ion-nav-view>

  </ion-side-menu-content>

  <!-- Menú lateral -->
  <ion-side-menu side="left">
    <!-- Título -->
    <ion-header-bar class="bar-stable">
      <h1 class="title">{{ 'menu.title' | translate }}</h1>
    </ion-header-bar>
    <!-- Contenido del menú -->
    <ion-content>
      <ion-list>
        <ion-item ng-show="getCurrentUserId() == null"
          menu-close ng-click="loadLoginPage()">
            <i class="icon ion-log-in"></i>
            {{ 'menu.login-item' | translate }}
        </ion-item>
```

Figura 78: Extracto de menu.html para Bilbozkatu

En la Figura 78 se muestra el uso de etiquetas Ionic como ***ion-side-menus***, ***ion-nav-bar***, o ***ion-content***, entre otros, para definir la barra superior y el listado de elementos del menú amén de otras configuraciones.

- ***"app.terms"***: el propio nombre indica ser un estado "hijo" del anterior, *app*, y por tanto una plantilla que se mostrará junto con la barra superior. En la Figura 77 se puede ver que también se configura la propiedad ***resolve***, con la que poder definir una función antes de abrir la pantalla de términos de uso. En este caso, tal y como se muestra en la Figura 79, se verifica si el usuario ya ha aceptado los términos y en tal caso se redirecciona a la pantalla de lista de propuestas.

```
/**
 * Code to be executed before route change goes to/app/terms
 */
TermsController.resolve = {
    checkBeforeDraw: function (UserLocalStorage, $ionicHistory, $state, $timeout, $q) {
        var isPrivacyAccepted = UserLocalStorage.getPrivacyAccepted();

        if (isPrivacyAccepted){
            goToProposalsList();
        }

        function goToProposalsList() {
            $timeout(function() {
                // Avoid back button in the next view
                $ionicHistory.nextViewOptions({ disableBack: true });
                $state.go('app.proposalslist');
            }, 0);
            return $q.reject();
        };
    }
};
```

Figura 79: Extracto de función previa carga de Términos de Uso

- **"*app.proposalsList*"**: este es tipo de estado típico generado también para el resto de pantallas. Corresponde a la pantalla de lista de propuestas, y con *AngularUI Router* se vuelve a definir la plantilla HTML y controlador asociados, además de la propia url con la posibilidad de definir parámetros. La definición de dicha plantilla, en este caso, se haría de la siguiente manera mediante las etiquetas ***ion-view***, ***ion-nav-title, ion-nav-buttons*** o la principal ***ion-content***, algunas de las cuales modifican la plantilla "padre" tal y como se ha comentado.

```
<ion-view>

  <!-- Título de la pantalla en la barra superior -->
  <ion-nav-title>
    {{ 'proposal-list-page.title' | translate }}
  </ion-nav-title>
  <!-- Botón para reordenar propuestas -->
  <ion-nav-buttons side="secondary">
    <i class="icon ion-android-more-vertical filters_show"
       ng-click="openPopover($event)"></i>
  </ion-nav-buttons>

  <ion-content class="fondo">
```

Figura 80: Extracto de plantilla para la lista de propuestas

En este caso se modifica la barra superior para mostrar el título de la pantalla, y para insertar un botón en la parte derecha con el que mostrar un panel para cambiar la ordenación de las propuestas mostradas. Como se ha dicho, ***ion-content*** incluye el grueso de la plantilla, haciendo uso de etiquetas HTML normales, elementos de Ionic y atributos de AngularJS para implementar cierta lógica.

Una vez que se ha analizado a grandes rasgos la implementación *Single Page* de la app con Ionic y Angular, se han podido seguir dos estrategias para la estructuración de los componentes del **MVC** de AngularJS. La primera, empleada en este caso, consiste en dividir los ficheros por tipo. Es decir, disponer por separado de los controladores, estados (*AngularJS Routing*) y plantillas utilizadas

para las pantallas de la app. La segunda estrategia consistiría en dividirlo por Casos de Uso, teniendo un directorio en el que se incluye el controlador, el estado y la plantilla por cada uno. Si bien las dos estrategias son válidas y se ha utilizado la primera, siguientes aplicaciones que están siendo desarrolladas externas al proyecto de fin de grado se están realizando con la segunda estrategia. A continuación se describen estos directorios (mostrados en la Figura 75), obviando *templates/* ya que sólo incluye todos los ficheros HTML correspondientes a las pantallas.

- ***controllers/***: reúne todos los controladores Angular asociados a todas las pantallas que se almacenan dentro del módulo ***bilbozkatuApp.controllers***, y en los cuales se inyectan los componentes o servicios definidos en Ionic y Angular para su uso (ver Figura 81).

```
angular
      .module('bilbozkatuApp.controllers')
      .controller('LoginCtrl', LoginController);

LoginController.$inject = ['$scope', '$state', '$ionicLoading', '$ionicPopup', '$filter', '$ionicHistory',
                           '$timeout', '$http', 'Login', 'UserLocalStorage', 'BILBOZKATU_BB_URL', 'KPI'];

/**
 * Controller - Use WeLve's AAC
 */
function LoginController($scope, $state, $ionicLoading, $ionicPopup, $filter, $ionicHistory, $timeout,
    $http, Login, UserLocalStorage, BILBOZKATU_BB_URL, KPI){

    /*
    * Properties
    */
```

Figura 81: Extracto de definición de controlador para la pantalla de Login

- ***directives/***: incluye las nuevas directivas AngularJS creadas, en este caso sólo una.
- ***services/***: se guardan los servicios (en Angular, "factorías") creados para su uso en los controladores, definidos para el módulo ***bilbozkatuApp.services***. Los servicios definidos son:
    - Un servicio para trabajar con las propuestas (almacenarlas y gestionarlas).
    - Un servicio para el proceso de autenticación.
    - Un servicio para gestionar el *LoggingBB*, es decir, con el que acceder y registrar los diferentes KPIs (logs) definidos para la aplicación.
    - Un servicio de acceso al *localStorage* de la app (memoria persistente interna de la app), a modo de capa de acceso a dichos datos.

  El modo de definición es muy similar al de los controladores (Figura 81), con la diferencia de que en vez de un *.controller*, lo que se registra es una *.factory* en el módulo ***bilbozkatuApp.services***.

- ***messages/***: incluye un fichero por cada idioma soportado (castellano, euskera e inglés en este caso). Dichos ficheros están en formato JSON, teniendo parámetros clave-valor para la internacionalización de la app.
- ***config/***: en este directorio se han definido dos ficheros de configuración relevantes en la app:
    - *categories***:** donde se definen las categorías en las que poder registrar propuestas, junto con el *path* de la imagen asociada a cada una.
    - *zones*: donde se definen las zonas o barrios en las que poder registrar propuestas, junto con las coordenadas geográficas de cada una para mostrar los marcadores correspondientes en la pantalla del mapa.
- ***app.module.js***: en este fichero se define el módulo principal de Angular de la app, ***bilbozkatuApp***, el cual se inyecta en el <*body/*> de *index.html* (ver Figura 76). En él se inyectan los módulos que se han ido creando (*bilbozkatuApp.controllers, bilbozkatuApp.services* y *bilbozkatuApp.directives*), además de otras dependencias de Ionic o de librerías JavaScript integradas. Tiene el método ***run*** el cual se ejecuta al iniciar la app (por ejemplo, se realiza la llamada al AAC BB para obtener el *token* de acceso de la app para hacer uso del *LogginBB*).
- ***app.state.js***: se define el módulo de configuración (*.config*(...)) de AngularJS, que es el primero que se ejecuta al arrancar la app. Por ejemplo, se configuran algunos proveedores como el de *angular-translate* para los idiomas (*$translateProvider*), el del componente de acceso al *localStorage* de la app para los datos persistentes en la memoria de la app (*$localStorageServiceProvider*), o el *$stateProvider* para definir todos los estados de la app (ver Figura 77).
- ***app.constants.js***: en este fichero se definen las constantes a utilizar. En este caso son las URLs de los *Building Blocks* desarrollados (*UsersFeedbackBB y BilbozkatuBB*) para poder cambiar rápidamente de utilizar los BB en local o en cualquier otro servidor. También se configuran otros parámetros como el número de *feedbacks* a obtener en la pantalla de detalles de una propuesta en cada bloque, o el intervalo de tiempo para comprobar si hay nuevos *feedbacks* o no. Por último, también se indica el periodo de validez de una propuesta desde la creación de la misma, estando inicialmente configurado a 2 meses hasta la expiración de la petición, cuando ésta cambia su estado a "cerrado".
- ***map.js***: contiene métodos auxiliares para hacer uso de la API de JavaScript de Google Maps, como la inicialización del propio mapa.

Plugins de Apache Cordova instalados

Si bien las librerías o plugins JavaScript (instalados mediante el gestor Bower) ya se han enumerado en el apartado correspondiente al análisis de Bilbozkatu, en el punto 6.3.1, a continuación se indican los *plugins* de Cordova:

- *cordova-plugin-device v1.0.1*
- *cordova-plugin-splashscreen 2.1.0*
- *cordova-plugin-whitelist v1.2.2*
- *cordova-plugin-inappbrowser v1.4.0*

En la fase de desarrollo también se había instalado el *plugin cordova-plugin-console*, pero al no necesitarse en la fase de producción se ha desinstalado antes de compilar la app para Android. Los dos últimos, whitelist e inAppBrowser han sido necesarios para la ejecución de un navegador nativo dentro de la propia app, tanto para el proceso de autenticación como para mostrar el cuestionario de opinión para el usuario. El resto de *plugins* venían instalados por defecto para diversos usos de Ionic.

Antes de proceder a analizar algún que otro aspecto concreto más de la implementación, cabe añadir que el uso de *frameworks* como Ionic y AngularJS han hecho que la app desarrollada presente una interfaz y navegación fluida y ágil (gracias también al cacheado de vistas), con transiciones y efectos que la hacen atractiva, además de ser una aplicación móvil robusta.

Proceso de autenticación de usuario

En relación a la implementación hay numerosos procesos y funcionalidades desarrolladas en las pantallas. No obstante, es interesante analizar un poco más en profundidad el proceso de autenticación de un usuario contra la plataforma de WeLive, aunque no sea de manera muy detallada, mediante el componente de WeLive AAC BB.

En apartados anteriores (5.2.1 y 6.3.2) se ha explicado dicho componente, y la necesidad de configurarlo en la consola web de desarrolladores de WeLive. En dicha consola se debe registrar la app en la que se vaya a integrar, y obtener algunos parámetros como el *clientId* y *clientSecret* para hacer uso de la API. También se configuran los proveedores de identidad que se mostrarán (Facebook, Google o WeLive) y los permisos del usuario que iniciará sesión en Bilbozkatu (en este caso sólo los de leer un perfil básico de usuario). Aunque no tiene que ver con los usuarios, también se debe configurar los permisos de la app para obtener el *token* de aplicación (distinto a *token* de usuario) para hacer uso del *LoggingBB*. Este concepto de *token* es debido al uso de *OAuth* en el proceso.

En la siguiente Figura 82 se muestra el flujo de eventos en el proceso de autenticación de un usuario utilizando la API de dicho componente AAC BB.

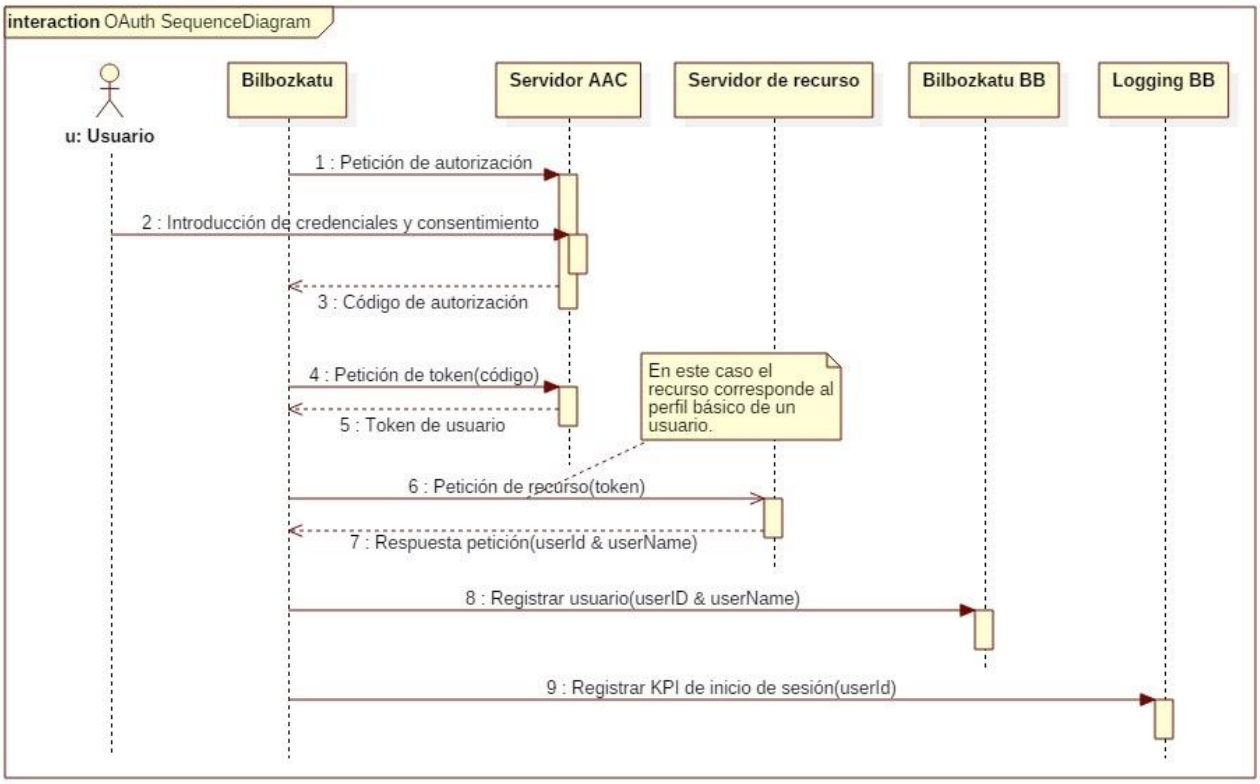


Figura 82: Diagrama de secuencia simplificado de login en Bilbozkatu

El diagrama no muestra las capas en las que el proceso está implementado en la app Bilbozkatu, que son el controlador de la pantalla de login y el servicio de login. Dicho servicio tiene implementadas las diferentes llamadas y demás a las APIs externas, junto con las configuraciones y validaciones correspondientes. Gracias a los elementos ***$q*** y ***$http*** de AngularJS se gestionan las llamadas **asíncronas**, de manera que el proceso continúa sólo si el paso previo ya ha terminado (de forma satisfactoria o error). Esto se muestra en la siguiente Figura 83.

```
Login.requestAuthorize()
    .then(function(){
        $ionicLoading.show({
            template: '<ion-spinner icon="bubbles"></ion-spinner><br/>'
                    + $filter('translate')('login.loading')
        });
        return Login.requestOauthToken();
    })
    .then(Login.requestOauthTokenSuccessCallback)
    .then(Login.requestBasicProfile)
    .then(requestBasicProfileSuccessCallback, showAACerrorMsg);
```

Figura 83: Proceso de login principal en el controlador de Login

En la anterior imagen se pueden ver claramente los puntos 1, 4 y 6 del diagrama de la Figura 82. En el método *requestAuthorize*() del servicio *Login* implementado se ejecuta en el navegador interno de la app el componente

gráfico del AAC BB (ver la Figura 84), en el que el usuario continúa el proceso de login tal y como ya se ha analizado en las interfaces correspondientes (en el punto 6.3.2.2). Después de que el usuario haya elegido un proveedor de identidad, introducido las credenciales y aceptado el consentimiento, el AAC BB devuelve el control a la app Bilbozkatu (gracias a un parámetro *callbackUrl* que la app queda a la espera de detectar) devolviendo el código de autorización.

```
var requestUrl = 'https://dev.welive.eu/aac/eauth/authorize'
    + '?'
    + 'client_id=' + login.params.clientId
    + '&'
    + 'response_type=' + login.params.responseType
    + '&'
    + 'redirect_uri=' + login.params.redirectUri
    + '&'
    + 'scope=' + login.params.scope
    + '&'
    + 'state=' + login.params.state;

// open a new window with the request URL
var ref = window.open(requestUrl, '_blank', 'location=no,clearcache=yes');
console.log('Start URL:' + requestUrl + ' END URL');

// set a listener in order to manage the loadstart event
ref.addEventListener('loadstart', loadStartListener);

// set a listener in order to manage the loaderror event
ref.addEventListener('loaderror', refuseLoginFlow);
```

Figura 84: Extracto de ejecución del AAC en el inAppBrowser en el servicio 'Login'

```
function loadStartListener(event) {
    // check if the url is the same of the redirection
    if ((event.url).indexOf("http://localhost/callback") > -1){
        // take the requestToken from the url
        login.code = (event.url).split('code=')[1].split('&')[0];
        // close the opened window
        ref.close();
        // unsuscribe event
        ref.removeEventListener('loadstart', loadStartListener);
        // resolve promise
        resolve();
    }
}
```

Figura 85: Extracto de detección y finalización del inAppBrowser en el servicio 'Login'

Dicho código se utiliza después en el método *Login.requestOAuthToken()* para obtener el *token* del usuario, y con él realizar una petición de su perfil básico, en el que se incluyen, entre otros, el *userId* y el *userName* (nombre) del usuario. Cabe añadir que en las pantallas del componente AAC BB, además de introducir las credenciales para iniciar la sesión, el usuario también tiene la oportunidad de registrar una nueva cuenta de WeLive. Si no, también es posible utilizar los proveedores de Google y Facebook para hacerlo, ya que así se ha configurado en esta aplicación.

En los siguientes dos extractos se muestran las funciones de los pasos 4 y 6 del diagrama, es decir, petición del *token* del usuario en base al código de autorización recibido, y acceso al perfil básico del mismo con dicho *token*.

```
/**
 * @desc (token generation) Request the OAuth token to get permission
 * to other requests of WeLive's APIs
 */
function requestOauthToken() {
    $http.defaults.headers
        .post['Content-Type'] = 'application/x-www-form-urlencoded';
    return $http({
        method: 'POST',
        url: 'https://dev.welive.eu/aac/oauth/token',
        data: 'client_id=' + login.params.clientId
            + '&'
            + 'client_secret=' + login.params.clientSecret
            + '&'
            + 'code=' + login.code
            + '&'
            + 'redirect_uri=' + login.params.redirectUri
            + '&'
            + 'grant_type=authorization_code'
    });
}
```

Figura 86: Extracto de petición de token de usuario

```
/**
 * @desc Get current user's basic profile (name, surname, socialId and userId)
 * @param token: previously generated token
 * (resolved by 'requestOauthTokenSuccessCallback()')
 */
function requestBasicProfile(token){
    return $http.get('https://dev.welive.eu/aac/basicprofile/me', {
        headers:{
            'Authorization': 'Bearer ' + token
        }
    });
}
```

Figura 87: Extracto de petición de perfil básico de usuario

Una vez obtenido el perfil básico del usuario, el flujo del proceso se sitúa en el controlador de la pantalla de login, en la cual se inició gracias a un botón. A continuación realiza tres cosas: registrar el *userId* y *userName* en el *BilbozkatuBB* (ya que es un servicio independiente con otra base de datos, y se deben guardar los usuarios para controlar los votos, etc., como se ha visto en el punto 6.2), registrar el KPI o log de inicio de sesión en el componente *LoggingBB* de WeLive y, finalmente, almacenar los datos del perfil del usuario en los datos internos de la app, de manera que si no se cierra la sesión al iniciar de nuevo la app tenga la sesión iniciada.

Si sucede esto último, la aplicación al arrancar realizará una llamada a la API del AAC de WeLive para refrescar el *token* del usuario, gracias al *refreshToken* obtenido también en el proceso de login con el que actualizar el *accessToken* del usuario. Y es que los *tokens* generados tienen un tiempo de vida limitado.

Usos de otros componentes o librerías

Como ya se ha ido comentando, tanto en las plantillas HTML como en los ficheros JavaScript (sobre todo en controladores y servicios) se han utilizado numerosos componentes de Ionic y AngularJS. Desde la navegación, pasando por condiciones para mostrar o no elementos HTML o generar dinámicamente elementos en HTML en base a una variable que es un *array*, hasta numerosos componentes JavaScript para obtener mensajes filtrados por idiomas, realizar llamadas AJAX, forzar ejecuciones secuenciales de llamadas asíncronas, mostrar paneles informativos predefinidos en Ionic, etc.

También ha habido que gestionar los logs definidos para Bilbozkatu, como se ha comentado anteriormente. En WeLive hay una serie de KPIs definidos para cada aplicación, como el inicio de sesión de un usuario, creación de una propuesta, etc. En cada acción definida se debe registrar un log en el *LoggingBB*, para lo cual hace falta un *accessToken* de aplicación, el cual no expira, ya que dicho *Building Block* está securizado.

El formato y parámetros a indicar en cada log está predefinido, de manera que se ha creado un servicio que gestione estas llamadas. Así, desde el controlador o servicio oportuno de la app sólo se debe ejecutar el método correspondiente, como se puede ver en la siguiente Figura 88 de generación del log cuando un usuario crea una nueva propuesta o petición.

```
// KPI id: KPI.BIO.9
function votingCampaignOrganized(votingCampaignID, votingCampaignName) {
    if (enabled && isTokenRequest) {
        return $http({
            method: requestParams.method,
            url: requestParams.url,
            headers: requestParams.headers,
            data: {
                "msg": "Voting campaign created [KPI.BIO.9]",
                "appId": appId,
                "type": "VotingCampaignOrganized",
                "custom_attr": {
                    "appname": appName,
                    "VotingCampaignID": votingCampaignID,
                    "VotingCampaignName": votingCampaignName
                }
            }
        });
    }
}
```

Figura 88: Extracto de ejemplo de llamada al LoggingBB

En el KPI de creación de una propuesta se debe registrar el id y nombre (título) de dicha petición, además de otros parámetros en formato JSON como se puede ver. En cuanto a la llamada AJAX se deben indicar el método (POST), la url del *LoggingBB* y algunos *headers* (*Authorization*, entre otros). Por otro lado el uso de la API Google Maps también ha requerido de estudio, aunque ya se tenía algo de experiencia en el uso de esta API.

Técnicas empleadas

Antes de proceder a comentar algunos problemas encontrados en el desarrollo de esta aplicación, cabe destacar que la implementación realizada se ha tratado de hacer aplicando algunas de las buenas prácticas en el uso de AngularJS. Aunque bien es cierto que alguna como el uso de alias para los controladores no se ha realizado por desconocimiento y falta de tiempo, aunque el autor del presente documento ya emplea de inicio todo lo aprendido en siguientes proyectos con AngularJS e Ionic.

También es importante señalar la técnica empleada para la visualización de *feedbacks* en la pantalla de detalles de una petición. Inicialmente, al implementar tanto dicha pantalla como el *Building Block UsersFeedbackBB*, se contemplaron varios escenarios. Por un lado, el obtener los *feedbacks* de más nuevo a más viejos por bloques. Y, por otro, la comprobación de existencia de nuevos *feedbacks*.

Así, se siguió una lógica que, a posteriori, se ha visto que es una técnica utilizada también en otros servicios llamada ***cursoring***. Y es que en siguientes proyectos con Ionic en los que el autor ha seguido trabajando, ha habido que hacer uso de la API de Twitter para visualizar un *timeline*. En la documentación de dicha API la obtención de *tweets* al parecer está implementada con esa técnica. Se definen tres parámetros para indicar a partir de qué *tweet* se desean obtener más *tweets*, desde cuál y cuántos. En *BilbozkatuBB* se han utilizado esos mismos tres parámetros para implementar la paginación que se emplea en Bilbozkatu, aunque en este caso los parámetros son fechas con un determinado formato (incluyendo la zona horaria ya que la base de datos de *BilbozkatuBB* en la nube está en otra zona horaria), y en la API de Twitter se utilizan los propios *id's* de los *tweets*, con la misma filosofía.

La paginación de propuestas es similar, además utilizando los *id's* de las mismas como parámetros, aunque en ese caso sólo se pueden obtener bloques más viejos ya que no se comprueban si hay nuevas propuestas de manera automática.

6.3.3.1. Problemas encontrados

Al igual que en los servicios web *UsersFeedbackBB* y *BilbozkatuBB*, uno de los mayores problemas en el desarrollo de este artefacto software ha sido el completo desconocimiento de las, tecnologías, herramientas, *frameworks*, plugins, librerías y conceptos a utilizar (Ionic, AngularJS, Cordova, gestores de paquetes como Bower y librerías como la de Google Maps, etc.). Ha sido necesaria una formación previa de muchas cosas, y una constante investigación y aprendizaje en el uso e integración de diferentes *frameworks*.

Por supuesto, también ha sido necesario estudiar e integrar los componentes de WeLive, teniendo éstos además cambios en su especificación a lo largo del

desarrollo del proyecto. Ha sido necesaria una comunicación con otras empresas que forman parte de WeLive y que han participado en dichos componentes, como Tecnalia o Deusto en España, o FBK en Italia.

En cuanto a la propia implementación, ha habido problemas con el uso del *inAppBrowser* o navegador interno de la app (para el proceso de login o mostrar el cuestionario de opinión al usuario). Ha pasado bastante tiempo hasta que finalmente se ha conseguido que funcione sin problemas en dispositivos Android, principalmente por problemas con el *plugin* de Cordova y su configuración.

Por otro lado, los cambios en las especificaciones de algunos componentes y el propio uso de los mismos ha tenido complejidad, como el proceso de autenticación con OAuth 2.0 mediante el AAC BB. Esto incluye también el refrescar el *token* del usuario cada vez que se inicia la app con la sesión iniciada, o gestionar el *token* de la app para que internamente durante la ejecución de la app registre en el *LoggingBB* los logs correspondientes.

6.3.4. **Verificación y pruebas**

El desarrollo de la app se ha realizado principalmente gracias a Ionic ejecutando > *ionic serve*. Gracias a esto se inicia un servidor en el que poder ejecutar la app en el navegador de la máquina donde se ha desarrollado, Google Chrome en este caso. Este servicio admite los llamados **cambios en caliente**, de manera que estando la app en el navegador cualquier cambio en el código se refleja automáticamente en la app.

De este modo se han ido realizando pruebas continuas a lo largo de la implementación de la aplicación, haciendo uso también de la consola de desarrolladores del navegador Google Chrome. No obstante, existen llamadas a componentes externos u otras cosas no admitidas en >*ionic serve*, de manera que puntualmente algunas cosas se han ido testeando instalando la app en dispositivos Android. Otras, evidentemente, sólo se podían probar en la app como el uso del *inAppBrowser* para el proceso de login.

Se ha hecho uso de la herramienta ***Swagger Interface*** explicada en el apartado 5.2.4 para probar llamadas directamente a las APIs de los componentes de WeLive, sin necesidad de este modo de gestionar *tokens* de usuario para acceder a recursos protegidos del sistema de WeLive.

Por último, es muy importante destacar que hacia el final del desarrollo del proyecto, **personal** de la empresa **Eurohelp** y de empresas externas como **Deusto** han testeado y probado la app. Esto es importante en el desarrollo de aplicaciones, ya que el desarrollador quiera o no tiene las pruebas a realizar en mente, y ejecutará las acciones del modo y en el orden que ya tiene interiorizado. De este modo, usuarios que no han tenido nada que ver con el desarrollo de la app han detectado algún que otro *bug* o error a arreglar, y sobre todo **han propuesto muchas mejoras** que se han terminado implementando.

6.3.5. **Publicación**

La aplicación se ha compilado para la plataforma Android para su publicación en la tienda Google Play, disponible para versiones 4.1+ y 4.1.1+ hasta la 6.0. Esto se ha hecho gracias a Apache Cordova, ya que a través del CLI se completado los siguientes pasos:

1. Ejecutar > *ionic platform add Android* para generar el código de la app para Android en base a lo configurado en *config.xml*.
2. Desinstalar *plugins* que no sirven en la fase de producción como *cordova-plugin-console* mediante la ejecución de > *cordova plugin rm cordova-plugin-console*.
3. Compilar la app para Android con > *cordova build --release Android.*
4. Utilizar la herramienta *keytool* para generar una clave privada con la que firmar la app (cifrado RSA).
    - Firmar, mediante la herramienta *jarsigner*, el .apk compilado en el punto 3 con la firma del punto 4, indicando el *passphrase* puesto al crear la firma. Esto se debe hacer como medida de seguridad para que sólo el desarrollador pueda modificarla, para distribuir la aplicación correctamente y porque es un requisito de Google Play.
5. Utilizar la herramienta *zipalign* de los *Android SDK Tools* para alinear el .apk y obtener el fichero **Bilbozkatu.apk** definitivo.
6. Aritz Rabadán, responsable del proyecto WeLive en la empresa Eurohelp, es el encargado de subir Bilbozkatu.apk a Google Play e indicar los datos correspondientes (descripción de la app, etc.).

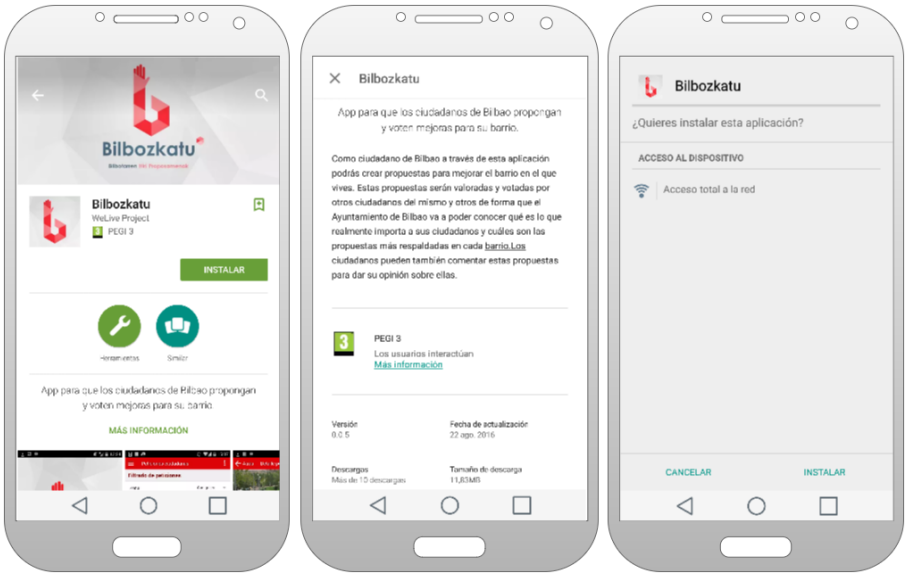


Figura 89: Bilbozkatu en Google Play y permisos de instalación

6.4. App BilbOn

Este es el último punto correspondiente al capítulo 6 de desarrollo del software, que trata la segunda de las apps híbridas desarrolladas dentro del proyecto WeLive, BilbOn. El modo de concebir esta app es el mismo que para Bilbozkatu de modo que, al igual que ha sucedido con el análisis de los *Building Blocks* en los apartados 6.1 y 6.2, muchas de las cosas que se deberían comentar para BilbOn son similares o iguales a las de Bilbozkatu y, por tanto, se obviarán las mismas explicaciones.

Figura 90: Logo App BilbOn

A continuación se explican los mismos puntos que para Bilbozkatu: análisis, diseño, algunos apuntes sobre la implementación, verificación y pruebas y, por último, la publicación de la app. En el punto 6.4.2.2 donde se analizan las pantallas se muestran las funcionalidades a modo de **manual del usuario**, a fin de facilitar el uso de esta aplicación móvil.

6.4.1. **Análisis**

En el punto 6.4.1.1 siguiente se describe de manera general el objetivo de la app y se enumeran los requisitos y funcionalidades que esta tiene. Muchos de los requisitos son los mismos que los de la app Bilbozkatu, de modo que como sucede a lo largo del punto 6.4 se referenciará a la sección de Bilbozkatu cuando corresponda.

6.4.1.1. Especificación y requisitos

BilbOn es una app móvil híbrida con la que los usuarios, ciudadanos de Bilbao, pueden buscar y filtrar diferentes Puntos de Interés o *Points of Interest* (en adelante, POIs) de la ciudad. Estos POIs se visualizan sobre un mapa y es posible acceder a sus detalles. Pueden ser tanto oficiales (ofrecidos por el Ayto. de Bilbao) como de ciudadanos, ya que se permite a los usuarios autenticados de la app crear nuevos puntos.

Al igual que en Bilbozkatu, es una app multilenguaje que soporta euskara, castellano e inglés, y dispone también de un cuestionario a través del cual los usuarios pueden valorar y dar su opinión sobre la app BilbOn.

Requisitos técnicos

Los requisitos técnicos en cuanto a las herramientas, *frameworks* y tecnologías a usar son los mismos que en Bilbozkatu, especificados en el punto 6.3.1.1, de modo que no se comentan de nuevo.

Requisitos funcionales

Seguidamente se enumeran las características que debe tener BilbOn, algunas de las cuales son similares o idénticas a las ya mencionadas para Bilbozkatu:

- Inicialmente se muestra un ***splash screen*** con los efectos, logotipos y textos correspondientes a WeLive, el Ayto. de Bilbao y la Comisión Europea.
- La primera vez se muestran los **Términos y Condiciones** de uso de la aplicación, que deben ser aceptados para acceder a la app.
- Tiene soporte **multilenguaje**: castellano, euskara e inglés.
- Dispone de una **pantalla inicial**, la principal, en la que se muestra un **mapa** sobre el que se mostrarán los marcadores correspondientes a los POIs encontrados.
    - Los **POIs** están catalogados en base a una serie de **categorías** predefinidas: "Restaurantes, sidrerías y bodegas", "Oficinas de turismo", "Alojamientos turísticos", "Paradas de taxi", "Farmacias" y "Museos".
    - Los POIs se visualizan mediante un **icono** que dependerá de la **categoría** a la que pertenece, además de si es **oficial** (del Ayto.) **o de ciudadano**. En este segundo caso el icono es el mismo pero con los colores invertidos.
- En todo momento se muestra una **barra superior** en la que se muestran los iconos para acceder a la pantalla de **inicio de sesión** y la pantalla ***About*** o de información y términos de uso.
- Sólo en la pantalla del mapa en la barra superior se muestra el icono para abrir el **menú lateral izquierdo**. Dicho menú contiene el **filtro** que se puede aplicar sobre los POIs y visualizar los resultados en el mapa. La búsqueda de POIs se puede realizar por:
    - La **categoría** a la que pertenecen los POIs.
    - La **ubicación geográfica** de los mismos en base a un radio predefinido de 500m. La ubicación se puede obtener:
        - Introduciendo un **texto** con el que se muestra una **lista de sugerencias de ubicaciones** en la que hay que seleccionar una de ellas gracias a Google Maps.
        - En base a la ubicación GPS del dispositivo. Si no está disponible, intentar obtenerla con la IP.
    - El **texto** que deben contener algunos parámetros (título, descripción...) de los detalles de los POIs.

    - Ver **también los POIs de ciudadanos**, ya que sin esta opción sólo comprueba los POIs oficiales del Ayto. de Bilbao.
    - Por último, por una **combinación de todos los anteriores** criterios para el filtro.
- En la pantalla del mapa el usuario puede **seleccionar** uno de los POIs encontrados pulsando el marcador correspondiente. Esto lleva al usuario a la pantalla de **detalles de un POI** en el que se **visualizan sus detalles**, a saber: nombre, descripción, categoría, página web, email, teléfono, localidad, provincia y tipo (oficial o de ciudadano). Es posible que algunos POIs no dispongan de todos los datos.
- Si el usuario está autenticado, a través del botón correspondiente de la pantalla del mapa puede abrir el formulario para **crear un nuevo POI** especificando algunos parámetros como el título, descripción o la localización.
- El usuario puede autenticarse a través de un botón en la barra superior: se hace uso del **componente AAC (*Authentication and Authorization Control System BB*) de WeLive** para la autenticación de usuarios.
- Dispone de una pantalla ***About*** en la que se muestra **información de la aplicación** y desde la que el usuario puede **ejecutar un cuestionario** sobre la app hasta un máximo de cuatro veces.
- Se hace uso del **componente *Logging BB* de WeLive** para registrar ciertos eventos en relación a la actividad de la aplicación, por ejemplo, "app iniciada" o "POI creado por el usuario x". Para ello la aplicación debe autenticarse también ante la plataforma WeLive (distinto a la autenticación del usuario, pero a través del mismo componente AAC BB).

Componentes externos

Generalmente los comentarios hechos en el apartado homónimo de Bilbozkatu son aplicables para BilbOn. En resumen, ha sido necesario un estudio en profundidad de los componentes de WeLive a integrar, librerías y plugins. Los componentes que se acceden e integran en la implementación de BilbOn son los siguientes:

- Plataforma WeLive
    - **AAC BB** para la autenticación de usuarios.
    - **LoggingBB** como servicio para registrar una serie de *logs* o KPIs definidos para la app.

- Componente para mostrar el **cuestionario** de la app (para registrar algunos datos por parte de los usuarios y conocer su opinión acerca de la utilidad y usabilidad de la app).
- Se hace uso del **componente ODS (*Open Data Stack*) de WeLive** para la consulta de POIs ofrecidos por el Ayto. de Bilbao (datos abiertos) y la creación de nuevos por parte de los ciudadanos. Este componente aborda los desafíos de acceder al conocimiento de una ciudad en forma de múltiples orígenes de datos en formatos diversos.

- Librerías JavaScript a través del gestor Bower

  Todas las librerías utilizadas son algunas de las ya mencionadas para Bilbozkatu. En concreto, las correspondientes de AngularJS para implementar el multilenguaje en base a ficheros y acceder al *localStorage* de la app, o la librería *ionic-modal-select* de Ionic, entre otras.

  No obstante, cabe destacar el uso de la **API de Google Maps v3**. Si bien en Bilbozkatu se utilizan algunas características para mostrar marcadores en un mapa, en BilbOn se hace un uso más exhaustivo de dicha librería, para lo que ha habido que estudiarla más a fondo. Por ejemplo, se ha utilizado la característica **Google Places** para implementar una búsqueda de ubicación en base a un texto que hace que se muestre una lista de sugerencias, y obtener después las coordenadas de la zona seleccionada. También se utiliza para saber si una coordenada está dentro de un radio predefinido respecto a otra, para realizar el filtro de ubicación.

6.4.2. **Diseño**

Siguiendo la estructura del punto 6.3 de la app Bilbozkatu, a continuación se procede a mostrar por medio de diagramas e imágenes el modelo de CU de BilbOn, así como el diseño de las distintas interfaces gráficas junto con una descripción de las mismas.

6.4.2.1. Modelo de Casos de Uso

La siguiente Figura 91 corresponde al modelo de CU de BilbOn. Algunos de ellos ya han aparecido en el modelo de Bilbozkatu como son "Cambiar de idioma", "Rellenar cuestionario de app", "Identificarse", y "Registrar usuario", las dos últimas haciendo uso del componente de WeLive AAC BB. El CU "Identificarse" extiende el de "Crear POI" porque, en el caso de que el usuario no esté autenticado, siempre se ejecuta. Ídem con "Registrar usuario", que se puede ejecutar desde dentro del AAC BB.

Por último, un usuario puede "Ver detalles de POI", para lo cual siempre hace falta que se ejecute también el CU "Filtrar POIs". Esto se indica mediante la anotación <<*include*>>, como se puede ver a continuación.

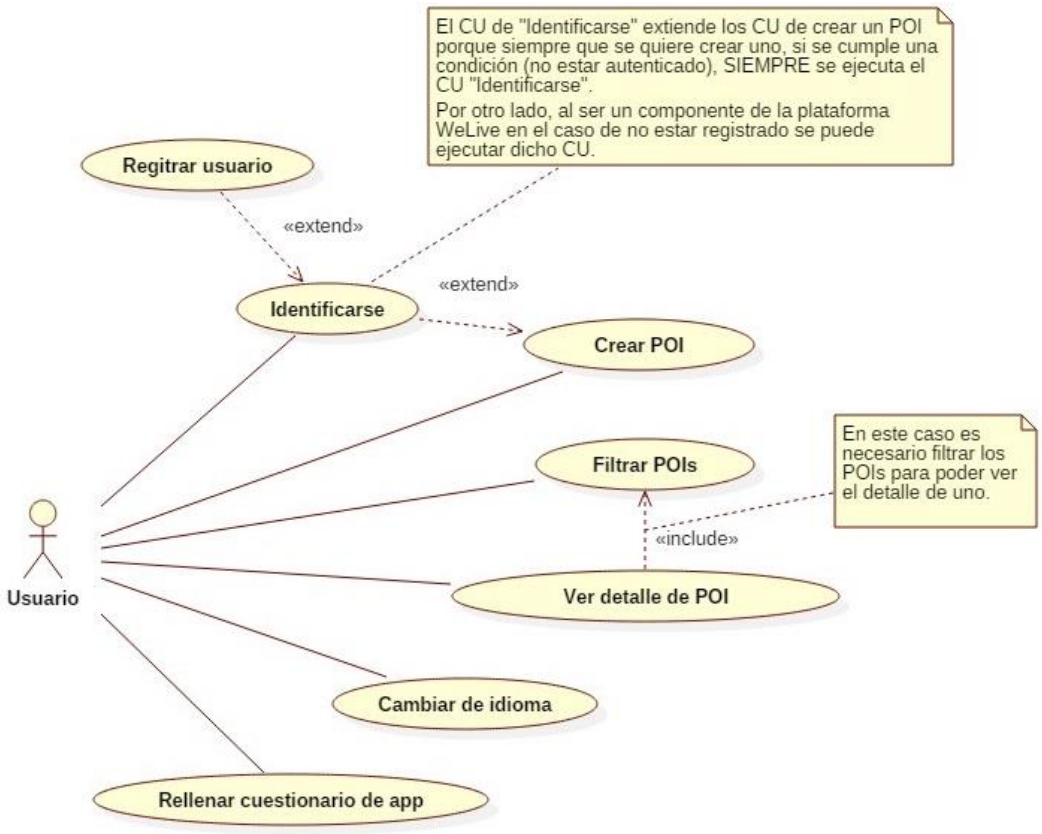


Figura 91: Modelo de Casos de Uso de la app BilbOn

6.4.2.2. Diseño de la interfaz gráfica

El diseño e implementación de las interfaces gráficas de la app parte de la base obtenida con Bilbozkatu. Se han utilizado los mismos tonos y estilos en la medida de lo posible, siendo el resto de elementos y estructura específicos de BilbOn. Y es que en esta app el menú lateral, como se mostrará a continuación, sólo está disponible en la pantalla principal de la app, la del mapa, y en él está implementado todo el filtro de POIs además de la selección del idioma.

Los accesos a la pantalla de inicio de sesión y de información y términos de uso se han reubicado en la barra superior. Dichas pantallas, por otro lado, son idénticas a las de Bilbozkatu, por lo que se han reutilizado las interfaces y lógica de las mismas. Cabe añadir que en un dispositivo se pueden visualizar los efectos en las transiciones al navegar por las distintas pantallas e interactiar.

Splash screen

Las dos pantallas iniciales del *splash screen* (la primera de las cuales es igual en Bilbozkatu) han vuelto a ser desarrolladas por **Alex Novoa**, diseñador gráfico de Eurohelp de Bilbao. En ellas se muestran algunos efectos en las transiciones, y mientras se muestran la aplicación comienza a cargarse y preparar la pantalla de inicio en un segundo plano. En la siguiente Figura 92 se muestran estas pantallas.

Figura 92: Pantalla splash screen de BilbOn

Términos y Condiciones de uso

Esta pantalla y su lógica es idéntica a la de Bilbozkatu, con el mismo objetivo (ver Figura 55 en el apartado 6.3.2.2). El usuario debe aceptar los términos de uso redactados por una comisión especializada del proyecto WeLive. Una vez aceptados se permite acceder a la app, y no se mostrará de nuevo a no ser que se eliminen manualmente los datos locales de la app.

Mapa

Esta es la pantalla principal de la aplicación. En ella se visualizan sobre el mapa los POIs obtenidos una vez aplicado el filtro. Dichos POIs se muestran como marcadores mediante iconos definidos en función de la categoría a la que pertenecen: "Restaurantes, sidrerías y bodegas", "Oficinas de turismo", "Alojamientos turísticos", "Paradas de taxi", "Farmacias" y "Museos". Por tanto, cada categoría tiene predefinida un icono, que además variará ligeramente en función de si es un POI oficial del Ayto. de Bilbao o es uno creado por un ciudadano a través de la app. Es decir, si por ejemplo un POI oficial de la categoría "Oficinas de turismo" tiene asociado un icono de fondo azul con una "i" blanca, un POI de dicha categoría creado por un usuario tendrá el mismo icono pero con los colores invertidos, diferenciando así la categoría y el tipo.

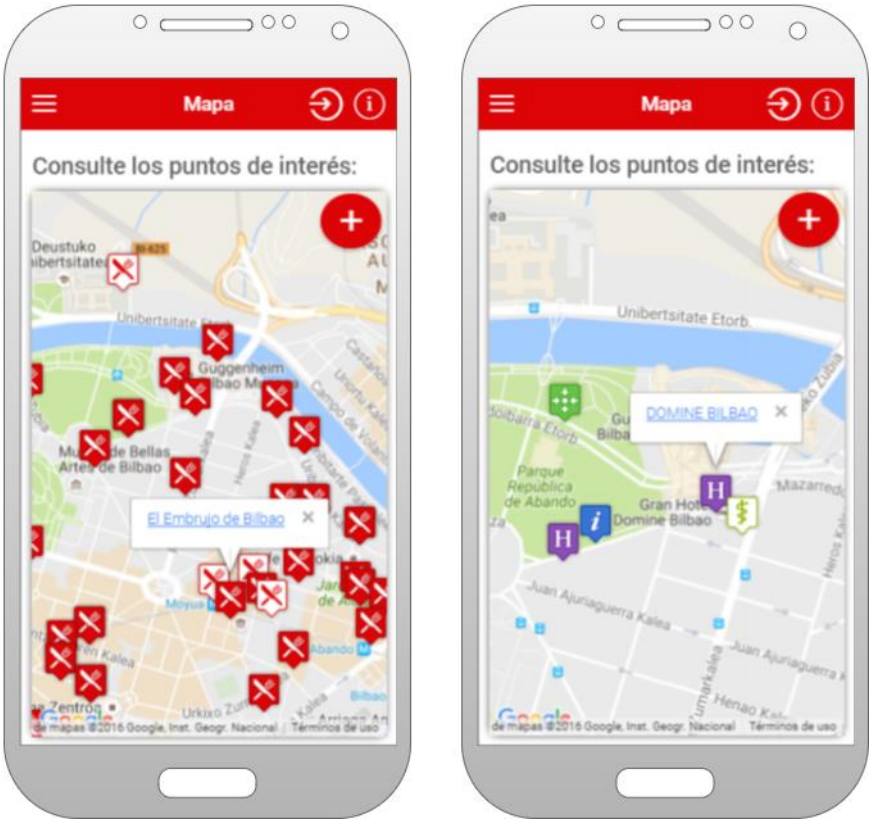


Figura 93: Ejemplos pantalla de mapa en BilbOn (1/2)

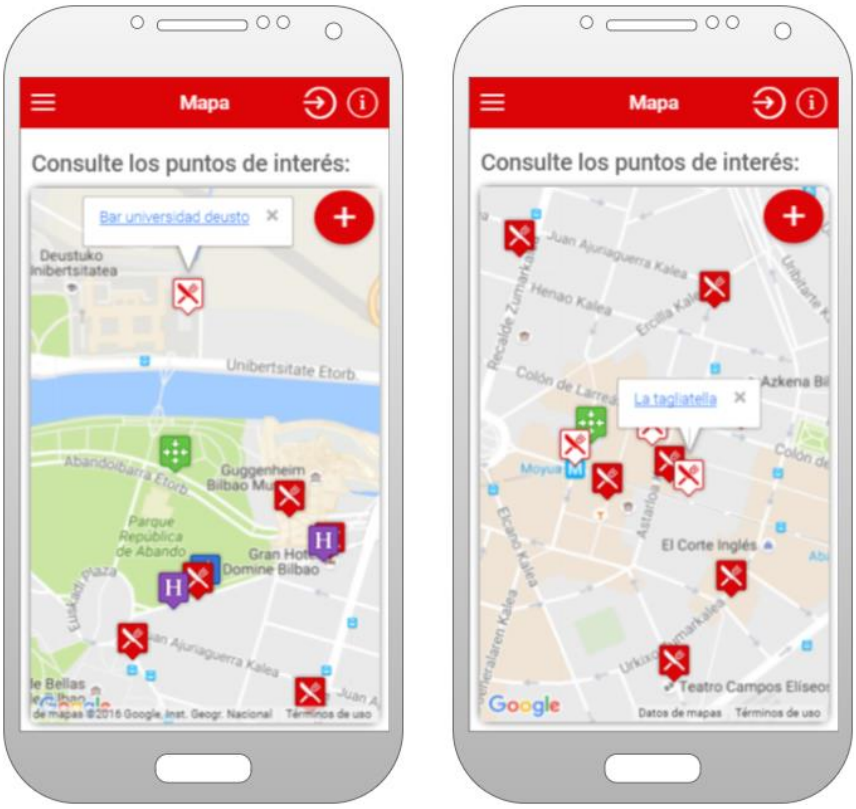


Figura 94: Ejemplos pantalla de mapa en BilbOn (2/2)

En las anteriores imágenes se muestran algunos ejemplos de los resultados obtenidos al aplicar el filtro con diferentes criterios. En la imagen izquierda de la Figura 93 se pueden ver los marcadores de los POIs de la categoría "Restaurantes, sidrerías y bodegas", incluyendo los POIs de ciudadanos (notar la inversión de colores en algunos iconos). Los iconos con fondo rojo corresponden a los POIs publicados por el Ayto. de Bilbao. Y al pulsar en un marcador aparece su nombre a través del cual se puede acceder a la pantalla de detalles.

En el segundo ejemplo de la Figura 93 el filtro es algo más complejo. En este caso las categorías que se muestran son "Alojamientos turísticos", "Oficinas de turismo" y "Farmacias". Además, se ha filtrado por ubicación, de manera que los POIs que se muestran son los que cumplen con el requisito de estar dentro de un radio predefinido de 500m con respecto a una ubicación geográfica indicada en el mapa con un icono verde de posicionamiento. En este ejemplo se muestran dos hoteles y una oficina de turismo de los datos oficiales, amén de una farmacia introducida por algún usuario.

En la Figura 94 se muestran más ejemplos, mezclando categorías y filtro de ubicaciones. Se puede observar también que en la barra superior se muestran los iconos con los que poder acceder a la pantalla de inicio de sesión y de información y términos de uso, además del icono para desplegar el menú lateral izquierdo correspondiente al filtro. El modo de uso y opciones de dicho filtro se analiza en el siguiente punto.

Por último, es interesante mencionar que cada vez que se cargan marcadores al mapa la visualización del mapa (posicionamiento y zoom) se acondiciona a los resultados obtenidos, mostrando un plano general si hay muchos POIs o centrándolo en una pequeña sección de Bilbao si sólo se han encontrado unos pocos, pero siempre mostrando todos.

Menú lateral, filtro

En las siguientes imágenes se pueden ver las distintas partes de las que se compone el filtro de POIs para su visualización en el mapa. El despliegue de dicho menú lateral se puede hacer tanto a través del icono de la barra superior como desplazando con el dedo dicho menú hacia la derecha.

El filtro se compone de los criterios "categoría", "ubicación", "texto" y "ver de ciudadanos". Por defecto siempre se muestran los POIs oficiales, a los que se pueden añadir los de ciudadanos con el parámetro comentado. Las categorías disponibles están predefinidas en la app, si bien es cierto que son fácilmente configurables (cosa que se comentará en la siguiente sección relativa a la implementación). El filtro de ubicación y de texto (cuyo texto se busca en los detalles de los POIs para aplicar el filtro) se aplica siempre sobre los POIs obtenidos inicialmente por cada categoría activada.

Evidentemente es posible combinar todos los parámetros de búsqueda disponibles, y dicho filtro se recarga cada vez que se cambia algún campo del mismo (se activan o desactivan filtros, o se cambia la ubicación o el texto con dichos filtros activados). En caso de no encontrar ningún POI se muestra un mensaje para informar de ello.

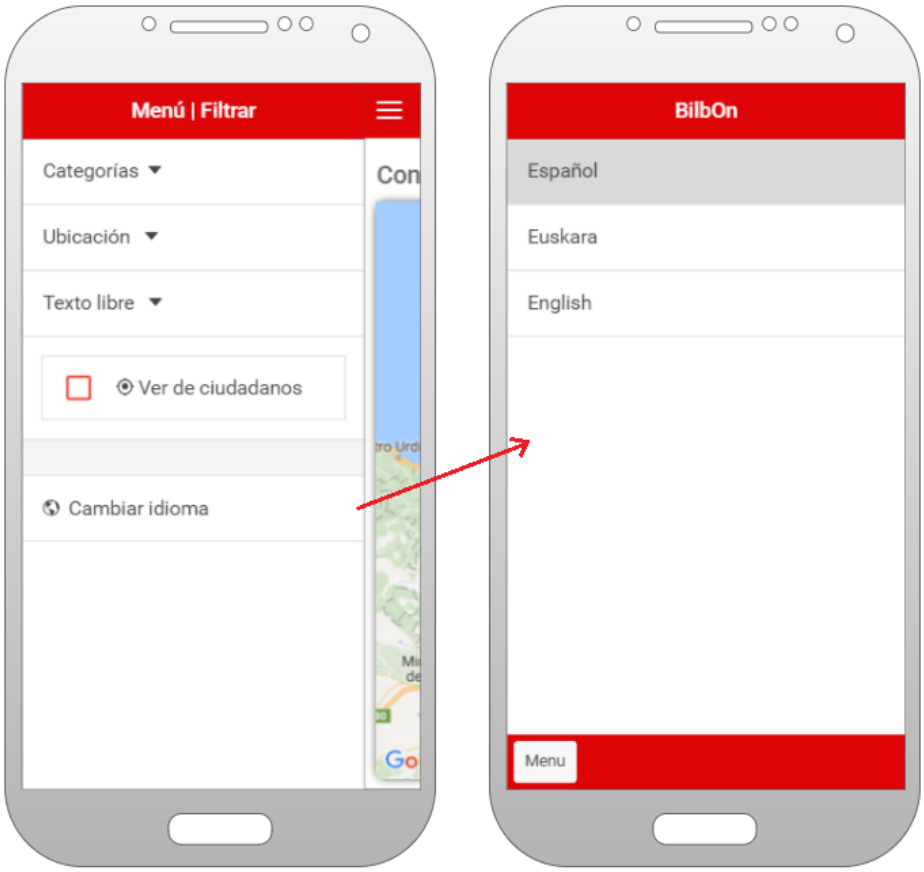


Figura 95: Filtro de POIs en elmenú lateral

Además del propio filtro, el menú lateral incluye la opción para mostrar el panel con el que poder cambiar de idioma, al igual que se hace en la app Bilbozkatu. Están disponibles el catellano, euskara e inglés, y es posible cerrar el panel sin seleccionar ningún idioma pulsando en el botón "Menú" que se muestra en la parte inferior izquierda del mismo.

En la siguiente Figura 95 se pueden ver las categorías disponibles en el filtro de categorías. También se muestra el desplegable para ver las opciones del campo de ubicación, que son el poder escribir una ubicación para elegir una de la lista de sugerencias, u obtener la posición GPS del dispositivo (o una posición aproximada en base a la dirección IP). Cada una de ellas hay que activarlas por separado según se quiera, pero son excluyentes (si se activa una, si la otra está activada se desactiva automáticamente). En el ejemplo se busca "Guggenheim" y se activa el filtro de ubicación/texto.

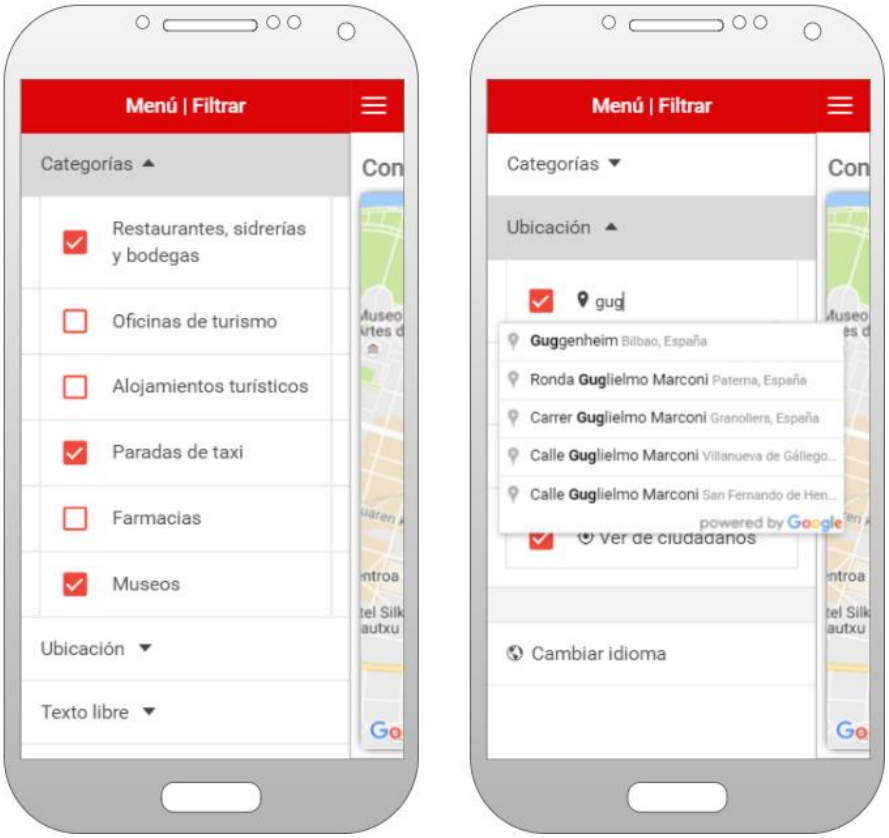


Figura 96: Ejemplos de filtro de POIs (1/2)

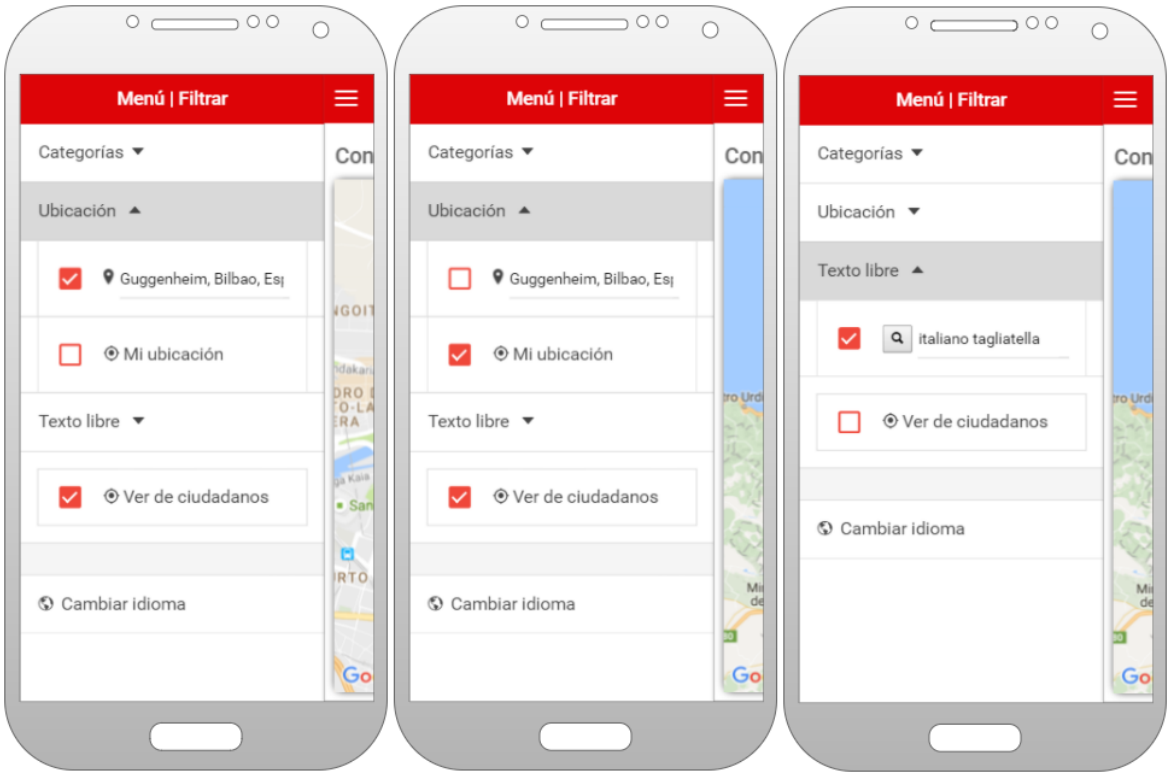


Figura 97: Ejemplos de filtro de POIs (2/2)

En cuanto al filtro de texto de la Figura 97 anterior, en la pantalla de la derecha se puede ver que se puede activar o desactivar el filtro, escribir un texto cualquiera para obtener los POIs en cuyos detalles debe aparecer, y pulsar un botón para actualizar el filtro en base a lo escrito. Se controlan cosas como que debe haberse escrito algo en la caja de texto, por ejemplo.

A la hora de activar los filtros de ubicación, texto o ver sólo de ciudadanos se comprueba que al menos una categoría haya sido activada. Si no es así se aborta el proceso y se informa al usuario. Tal y como se ha dicho anteriormente, el filtro de categorías es la base sobre la que se aplican el resto de criterios de búsqueda. Mensajes informativos que se muestran al realizar el proceso del filtrado o al no obtener resultados se pueden ver en la siguiente Figura 98.

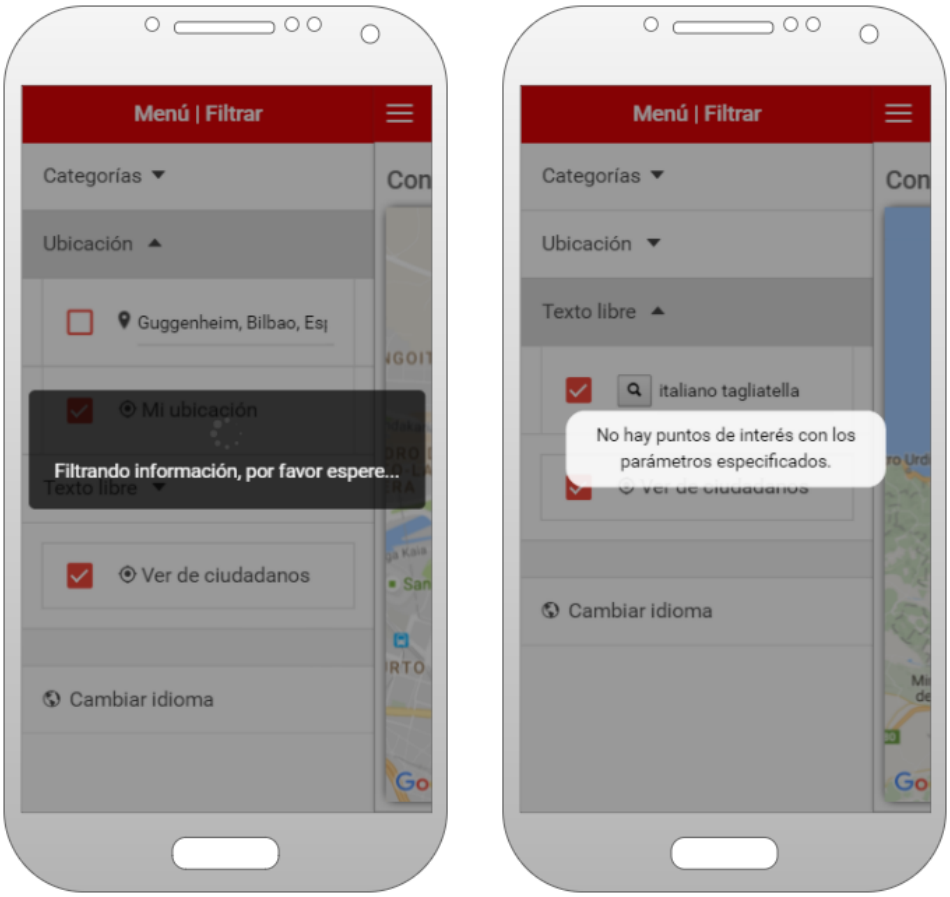


Figura 98: Ejemplos de mensajes del proceso de búsqueda de POIs

Detalles de un POI

Pulsando sobre el nombre del POI que se muestra al pulsar en el marcador del mismo en el mapa, se accede a la pantalla en la que se detallan los datos del POI. Algunos de los datos pueden no existir, y en tal caso se muestra un simple "-". A continuación se muestran los detalles de un POI oficial (el hotel Miró) y de uno creado por un ciudadano (restaurante La Tagliatella) de Bilbao.

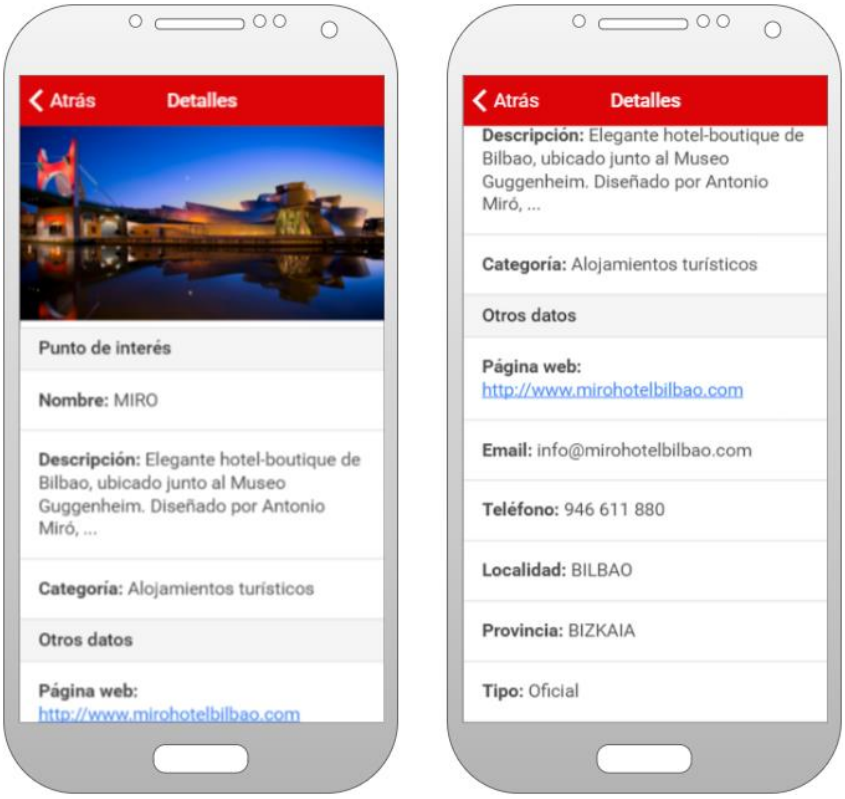


Figura 99: Ejemplo detalles de POI oficial

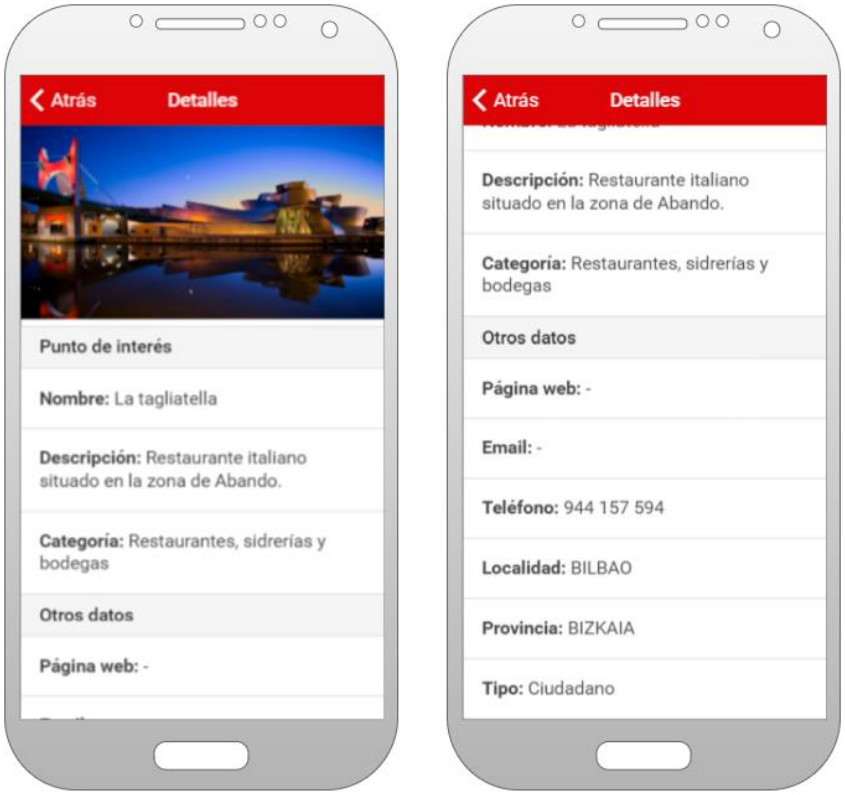


Figura 100: Ejemplo detalles de POI de ciudadano

Inicio de sesión

El inicio de sesión se ha implementado de igual manera a Bilbozkatu, de manera que se obvian las interfaces y explicaciones pertinentes al respecto. Tan sólo se muestra la propia pantalla a la que se accede con el icono del menú superior de la pantalla de login, y desde la que a través de un botón se inicia el proceso de login contra el AAC BB de WeLive tal y como ya se ha explicado.

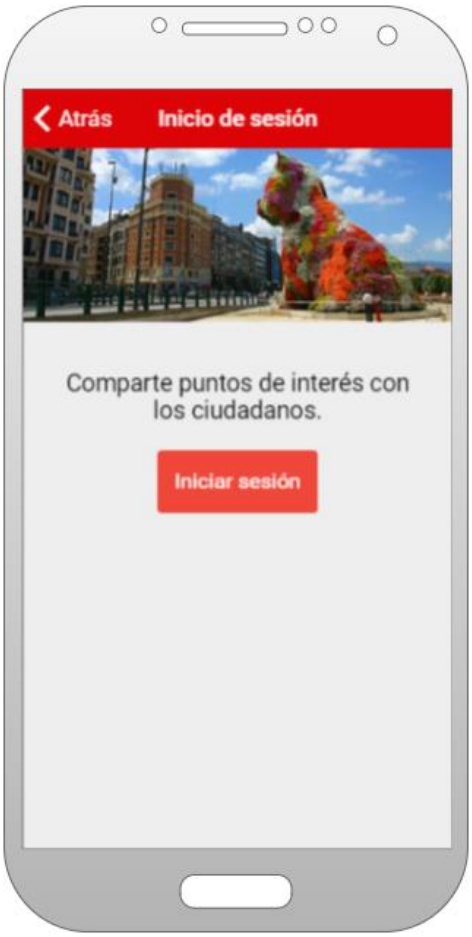


Figura 101: Pantalla de inicio de sesión de BilbOn

Crear nuevo POI

Una vez el usuario está autenticado tiene la posibilidad de crear un nuevo POI. La pantalla con el formulario de creación de POIs está accesible a través del botón con el símbolo "+" que está en la parte superior derecha del mapa (ver la Figura 93). En caso de no estar autenticado se muestra el mensaje informativo junto con un enlace directo a la pantalla de inicio de sesión, al igual que se hace en la app de Bilbozkatu.

El formulario de creación de un POI se compone de tres secciones. En la primera se deben indicar el "Nombre", "Descripción" y "Categoría" del POI (esta última ofrece como opciones las categorías predefinidas), siendo todos estos campos obligatorios.

En la segunda sección se pueden introducir otros datos que son opcionales: "Página web", "Email" y "Nº de teléfono". Por último se debe indicar, de manera obligatoria al igual que los datos de la primera sección, la ubicación del nuevo POI para poder mostrarlo en el mapa.

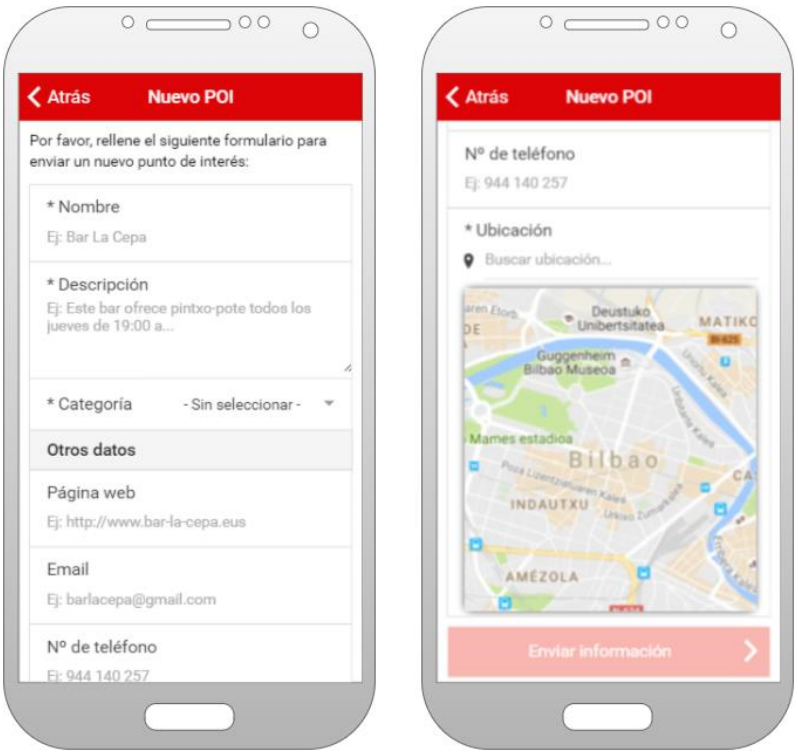


Figura 102: Formulario de creación de POI

Se realiza una validación exhaustiva de los campos del formulario y se muestran los mensajes de error correspondientes debajo de cada campo que los tiene. Hasta que no esté validado todo el formulario no es posible pulsar el botón de envío de datos. Así, por un lado, se controla que los datos obligatorios (nombre, descripción, categoría y ubicación) hayan sido introducidos. Y, por otro lado, se realizan las demás validaciones según el campo.

En el campo de la página web se valida que el texto introducido corresponda con el patrón de una dirección web. Además, como ayuda al usuario, en caso de no introducir el texto inicial "*http://*", éste se inserta automáticamente. De igual manera se valida que el email introducido tenga un formato correcto también, así como el número de teléfono que debe tener 9 dígitos. También se puede especificar un prefijo con los formatos '+34' o '0034', pero esto es opcional.

Por último, en cuanto a la ubicación, se valida que se hayan podido obtener las coordenadas. El usuario para ello puede escribir una ubicación y seleccionar una de la lista de sugerencias, o pulsar directamente sobre el mapa. En caso de producirse algún error se informa al usuario que vuelva a intentarlo, y en caso de éxito al obtener las coordenadas se visualiza sobre dicho mapa la localización señalada.

Por último, en cuanto a la ubicación se realiza otra validación más. No se acepta la ubicación si está fuera del rango de la ciudad de Bilbao, ya que los POIs creados sólo son válidos si están ubicados en Bilbao. En la siguiente imagen se puede ver este y los demás mensajes de errores definidos para cada campo del formulario.

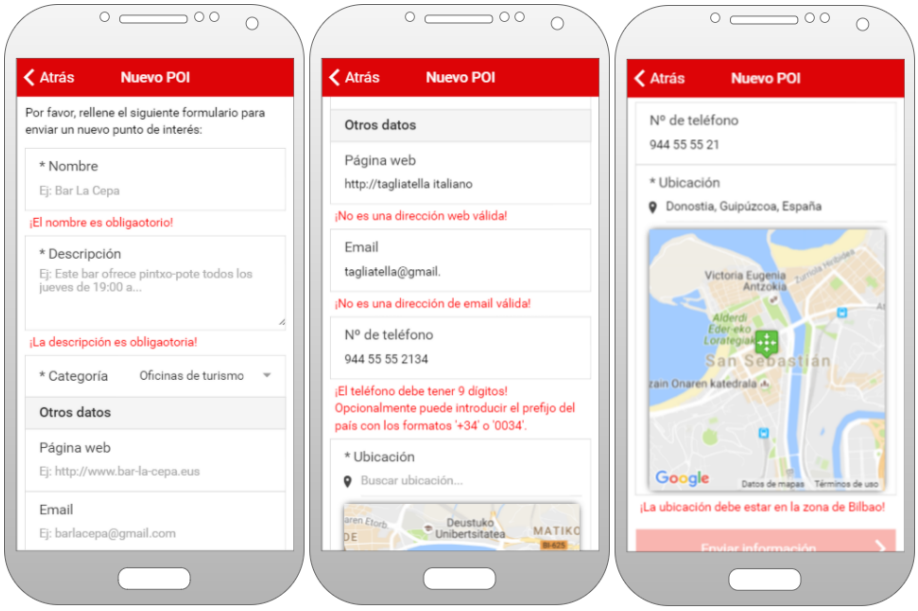


Figura 103: Validaciones del formulario de creación de POI

En la siguiente Figura 104 se puede ver la correcta introducción de una ubicación mediante texto, seleccionándola de la lista de sugerencias proporcionada por Google Maps. Una vez seleccionada la ubicación se almacenan las coordenadas y se muestra mediante un marcador en el mapa la ubicación seleccionada.

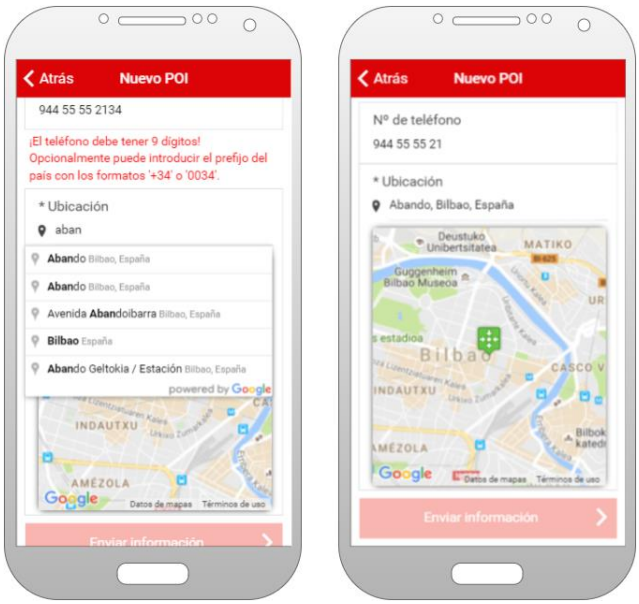


Figura 104: Introducción de ubicación en formulario de creación de POI

Información y términos de uso

Por último está la pantalla de información y términos de uso, prácticamente igual a la de Bilbozkatu excepto por el propio texto que se muestra. Nuevamente se muestra el botón "Rellenar el formulario" con el que poder ejecutar el cuestionario de opinión del usuario (hasta un máximo de 4 veces) sobre la utilidad y usabilidad de la app. Dicho cuestionario se ejecuta en el *inAppBrowser* o navegador interno de la app, y es un componente externo realizado en WeLive. Se puede ver en la Figura 73 del apartado de diseño de la app Bilbozkatu, en la sección 6.3.2.2.

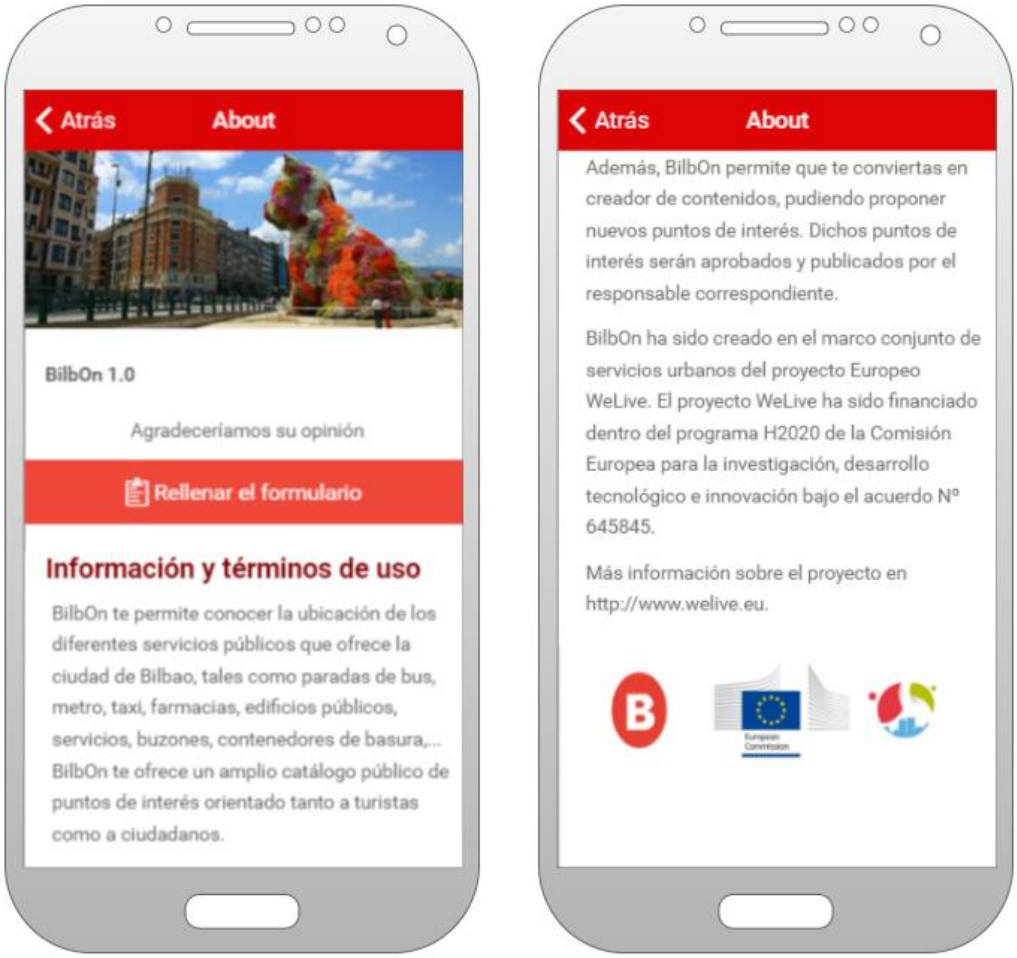


Figura 105: Pantalla de About de BilbOn

6.4.3. **Implementación**

En cuanto a la implementación se debe comentar que la aplicación, en cuanto a arquitectura, *frameworks*, tecnologías, librerías, componente WeLive AAC BB y *LoggingBB*, así como la propia estructura de ficheros y estrategia seguida para su definición es igual a la de Bilbozkatu. Por tanto es aplicable todo lo mencionado en el apartado relativo a la implementación de dicha app, en el punto 6.3.3.

En consecuencia, no hace falta comentar de nuevo la arquitectura general, la estructura de ficheros o los plugins utilizados. También se realiza de igual manera el proceso de inicio de sesión mediante el AAC BB y la gestión de *tokens* de usuario o de app, o el uso del *LoggingBB* para registrar los logs definidos par BilbOn (como el de "POIs buscados" o "accedido a detalles de POI").

No obstante, sí conviene mencionar algunos aspectos específicos e importantes de BilbOn, como son el uso de un nuevo componente de WeLive (el ODS, *Open Data Stack*), la instalación de un nuevo *plugin* de Cordova, el uso más exhaustivo de la API para JavaScript de Google Maps o la implementación del proceso de filtrado de POIs que ha sido la parte más compleja de programar en las dos apps además del proceso de login o el uso e integración de algunos componentes externos.

Componente *Open Data Stack* de WeLive

Este componente de WeLive se ha tratado en el apartado 5.2.3. En resumen, trata con los desafíos asociados a la gestión del conocimiento de una ciudad en la forma de múltiples orígenes de datos con distintos formatos. En relación a la app de BilbOn, se hace uso de su API para acceder de manera uniforme a distintos *datasets* de dicho componente.

Como se ha explicado, los POIs proporcionados por el Ayto. de Bilbao están divididos en categorías, lo que se traduce en que en el ODS existe un *dataset* por cada categoría. En realidad, en cada *dataset* hay definidos distintos *resources* que son ficheros en formato JSON, y por tanto el *dataset* de "Restaurantes, sidrerías y bodegas" tiene el *resource* que se utiliza en la app.

Utilizando la API del ODS, con una llamada a la misma se pueden consultar los POIs de un *resource* de un *dataset* en concreto, es decir, de una categoría, indicando como parámetro la sentencia SQL. Hay que tener en cuenta que el ODS trabaja internamente con una base de datos SQLite que realiza el mapeo a JSON. Así, las sentencias SQL deben ser correctas para dicho tipo de base de datos. En BilbOn esta consulta incluye los datos necesarios que se quieren obtener de los POIs, y en caso de aplicar también el filtro de texto se añade una cláusula WHERE para ello. Se puede acceder a la estructura de la tabla para poder realizar las sentencias mencionadas.

La estructura de las tablas que contienen POIs oficiales es igual. Sin embargo, los POIs de ciudadanos se gestionan de manera diferente ya que están todos almacenados en el mismo *dataset*, teniendo por tanto en cada uno un campo extra para especificar la categoría a la que pertenecen. Eso es algo a gestionar en el proceso de filtro, como se comentará después.

A continuación se muestra un pequeño extracto para el acceso al ODS, tanto para la consulta de POIs de una categoría (y con filtro de texto si hiciera falta) como para la creación de un nuevo POI por parte de un ciudadano.

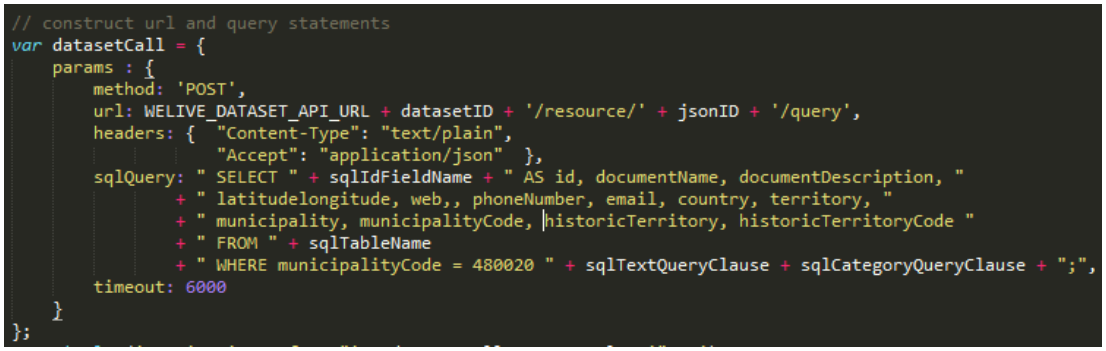


Figura 106: Extracto de definición de parámetros para consulta al ODS de WeLive

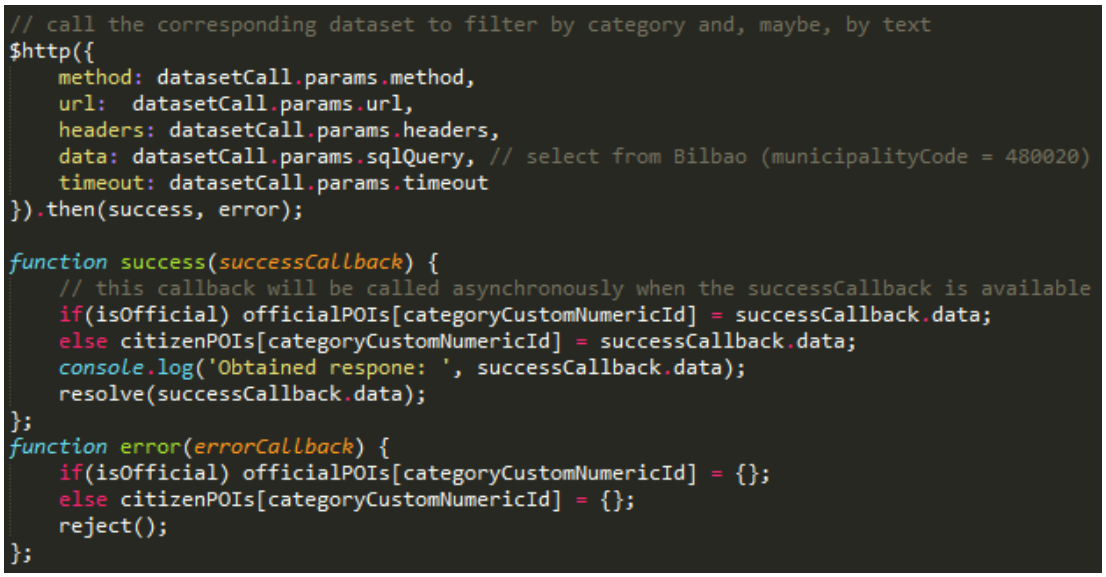


Figura 107: Extracto de llamada para consulta al ODS de WeLive

En las dos imágenes anteriores se puede ver cómo se preparan los parámetros a utilizar en el objeto *$http* de AngularJS para realizar una llamada AJAX al ODS de WeLive. En el parámetro *datasetCall.params.url* se indica la url de la API del ODS (*https://dev.welive.eu/dev/api/ods/*), el identificador del *dataset* y el identificador del *resource* (*jsonID* en el extracto de la Figura 106).

Por otro lado, en la Figura 107 se puede ver la propia llamada y la gestión de la respuesta. Como se puede ver, se hace uso de *resolve(...)* y *reject()*, lo que indica que el extracto está contenido en una función que devuelve el *promise* del objeto *$q* de AngularJS, que sirve para que la función que llama a esta detecte cuándo ha terminado su ejecución de manera satisfactoria o no. Esto ha sido una constante en el proceso del filtrado como se comentará más adelante.

Para terminar, a continuación se muestra otro extracto que pertenece a la creación de un nuevo POI por parte de un ciudadano, en el *dataset* reservado para los POIs de ciudadanos del ODS. Es necesario enviar el *token* del usuario para hacer uso del método */update* de la API, y por otro lado se envía la sentencia SQL con los datos del formulario.

```
// 'transaction' parameter of the URL: enable transaction mode for multiple update execution
var datasetCall = {
    params : {
        method: 'POST',
        url: WELIVE_DATASET_API_URL + datasetID + '/resource/' + jsonID + '/update?transaction=false',
        headers: {  'Content-Type': 'text/plain',
                    'Accept': 'application/json',
                    'Authorization': 'Bearer ' + UserLocalStorage.getAccessToken()  },
        sqlStatement: "INSERT INTO POIS (id, documentName, documentDescription, web, email, phoneNumber,"
                    + " latitudelongitude, category, municipalityCode, municipality, historicTerritoryCode,"
                    + " historicTerritory, country, territory) VALUES"
                    + " (null, '" + $scope.newPOI.documentName + "', '" + $scope.newPOI.documentDescription + "',"
                    + " '" + $scope.newPOI.web + "', '" + $scope.newPOI.email + "', '" + $scope.newPOI.phoneNumber + "',"
                    + " '" + $scope.newPOI.latitudelongitude + "', '" + $scope.newPOI.category + "',"
                    + " " + $scope.newPOI.municipalityCode + ", '" + $scope.newPOI.municipality + "',"
                    + " " + $scope.newPOI.historicTerritoryCode + ", '" + $scope.newPOI.historicTerritory + "',"
                    + " '" + $scope.newPOI.country + "', '" + $scope.newPOI.territory + "');",
        timeout: 7000
    }
};

// call the corresponding dataset to filter by category and, maybe, by text
return $http({
        method: datasetCall.params.method,
        url:  datasetCall.params.url,
        headers: datasetCall.params.headers,
        data: datasetCall.params.sqlStatement,
        timeout: datasetCall.params.timeout
});
```

Figura 108: Extracto de llamada de creación de POI al ODS

Plugin *Geolocation* de Apache Cordova

En BilbOn se ha hecho uso de un *plugin* de Cordova que no se había instalado en Bilbozkatu. Se trata de *cordova-plugin-geolocation* y sirve para acceder a la característica de la ubicación del dispositivo si está activada. Si en el dispositivo está activada la precisión se obtienen las coordenadas del mismo mediante GPS, de lo contrario se triangula la posición en base a la dirección IP de las redes móviles o wifi.

EL *plugin* es utilizado en el filtro de POIs por ubicación del dispositivo (no ubicación buscada por texto). En la siguiente Figura 109 se muestra el extracto en el que se intentan obtener las coordenadas del dispositivo para posteriormente poder aplicar el filtro de ubicación sobre los POIs que previamente se habían obtenido de la categoría correspondiente.

```
// get device's gps location
getDeviceLocation()
.then(function(positionCoords){ // success getting device's location
    // here, e.g., we also have 'positionCoords.accuracy'
    applyLocationFilter(categoryCustomNumericId, isOfficial, positionCoords.latitude, positionCoords.longitude)
    .then(function(filteredArray){ // success filtering
        resolve(filteredArray); // resolve 'callSelectedLocationFilter()'s promise
    }, function(){
        // the POIs array is null
        reject('bounds-error'); // reject 'callSelectedLocationFilter()'s promise
    });

}, function(){ // error getting device's location
    reject('gps-error'); // reject 'callSelectedLocationFilter()'s promise
});
```

Figura 109: Extracto de aplicación de filtro con coordenadas GPS

De nuevo se ha utilizado el elemento *$q* de AngularJS para poder definir funciones a ejecutar siempre de manera secuencial, tanto si hay error como si no. De modo que *getDeviceLocation()* devuelve un *promise*, y se especifican dos funciones en la sección *.then(...)*, una en caso de que *getDeviceLocation()* envíe

una respuesta positiva, y la otra si envía una respuesta de error. Dicha función se puede ver en el siguiente extracto.

```
// will execute when device is ready, or immediately if the device is already ready.
var options = {
  enableHighAccuracy: true,
  timeout: 10000,
  maximumAge: 0
};
function success(position){
    console.log('Device location: ', position.coords.latitude, position.coords.longitude);

    // store Google Places' coordinates to show the corresponding marker of the location filter
    setPositionFilterCoords(position.coords.latitude, position.coords.longitude);
    setPositionFilterCoordsStored(true);

    resolve(position.coords);
};
function error(error){
    // error.code = error.PERMISSION_DENIED | POSITION_UNAVAILABLE | TIMEOUT | UNKNOWN_ERROR
    reject();
};
// Try HTML5 geolocation.
if ("geolocation" in navigator) { // Check if Geolocation is supported (also with 'navigator.geolocation')
    // get device's location using GPS, IP or Wifi ('location' must be enabled on the device)
    navigator.geolocation.getCurrentPosition(success, error, options);
}else{
  console.log('geolocaiton IS NOT available');
  reject();
}
```

Figura 110: Extracto de getDeviceLocation() para obtener la ubicación

El uso del *plugin* se realiza en la instrucción *navigator.geolocation .getCurrentPosition(...)*. En caso de haber podido obtener las coordenadas GPS (se ha especificado en la variable *options* que *enableHighAccuracy* estuviera a *true*, es decir, que tratara de obtener la ubicación mediante GPS), se ejecuta el *resolve(position.coords)* que se detectará en el extracto de la Figura 109, ejecutando la primera función (f1) con la que se lanza *applyLocationFilter(...) .then(f1, f2)*.

Uso de la API JavaScript de Google Maps v3

Otro punto a destacar en cuanto a la implementación de BilbOn es el uso de la API de Google Maps. Esto es porque, a pesar de que también se había utilizado en Bilbozkatu, en este caso se han hecho uso de más características para lo que se ha requerido una investigación más profunda en el uso de la API como, por ejemplo, de Google Places. La **librería *places*** se debe añadir explícitamente a la hora de incluir la API de JavaScript de Google Maps en el fichero *index.html*, como se puede ver en el siguiente extracto.

```
<script type="text/javascript"
        src="https://maps.googleapis.com/maps/api/js?language=es&amp;libraries=places">
</script>
```

Figura 111: Extracto de inserción de Google Maps en index.html

Una vez hecho esto, en el controlador de la pantalla del Mapa se genera el objeto *google.maps.Map* para asociarlo al elemento del DOM en el que se quiere mostrar (un * *) el mapa, y el objeto *google.maps.places.Autocomplete* para

asociarlo al elemento del DOM (caja de texto) en el que se debe mostrar la lista de sugerencias de ubicaciones en base al texto escrito en dicha caja de texto ().

```
// initialize map and filter's Google Autocomplete's object
var map = initializeMap(document.getElementById('mapa'));
Map.setMap(map);

var autocomplete = loadGooglePlacesAutocompleteFeature(
    document.getElementById('location-input'),
    Map, $ionicPopup);
Map.setAutocomplete(autocomplete);
```

Figura 112: Extracto de carga de objetos Map y Autocomplete de Google Maps

En la Figura 112 se muestra cómo al inicializar la pantalla del mapa, a través del controlador, se generan los objetos *Map* y *Autocomplete* asociándolos a los elementos del DOM ya existentes tal y como se ha comentado. Una vez inicializados y asociados al HTML se almacenan sus referencias en un servicio de AngularJS que se ha creado con el nombre de '*Map*' para centralizar todas las gestiones con todo lo relacionado con dichos objetos.

En las siguientes imágenes se muestran las implementaciones de *initializeMap(...)* y *loadGooglePlacesAutocompleteFeature(...)*. En el primer caso se definen una serie de opciones para inicializar el mapa en Bilbao con una serie de parámetros como el zoom o el tipo de mapa a mostrar (en cuanto a estilo) ya definidos. Por otro lado, en el segundo caso se configura la característica de obtener una lista de sugerencias en base a un texto escrito, especificando un área geográfica a la que dar prioridad al mostrar dichas sugerencias que en este caso es la zona de Bilbao.

```
/**
*   Create map object with the corresponding parameters
*/
function initializeMap(domMapContainer){

    //var infoWindow = new google.maps.InfoWindow();
    var options = {
        zoom: 14,
        zoomControl: false,
        streetViewControl: false,
        mapTypeControl: false,
        center: new google.maps.LatLng(43.263606, -2.935214), // Plaza de Don Federico Moyúa, Bilbao
        mapTypeId: google.maps.MapTypeId.MAP
    };

    var map = new google.maps.Map(domMapContainer, options);

    google.maps.event.addDomListener(window, "resize", function() {
        var center = map.getCenter();
        google.maps.event.trigger(map, "resize");
        map.setCenter(center);
    });

    return map;
};
```

Figura 113: Extracto de creación de objeto Map de Google Maps

```
function loadGooglePlacesAutocompleteFeature(domInputElement, MapFactory, $ionicPopup){

    var bilbaoBounds = new google.maps.LatLngBounds(
        new google.maps.LatLng(43.199927, -3.017116),   //south-west corner
        new google.maps.LatLng(43.310109, -2.827070));  //north-east corner
    // (Google Places) Create the autocomplete object, restricting the search to geographical location types.
    var autocompleteObj = new google.maps.places.Autocomplete(
        /** @type {HTMLInputElement} */
        (domInputElement), {
            types : [ 'geocode' ],
            componentRestrictions: { country: 'es' },
            bounds: bilbaoBounds //The area in which to search for places.
                // Results are biased towards, but not restricted to, places contained within these bounds.
        });

    google.maps.event.addListener(autocompleteObj, 'place_changed', placeChangedListener);

    return autocompleteObj;
}
```

Figura 114: Extracto de creación de objeto Autocomplete de Google Maps

En la anterior Figura 114 se puede apreciar la creación de un *listener*, es decir, la definición de la función *placeChangedListener(...)* como método a ejecutar cuando el objeto *google.maps.places.Autocomplete* detecta un cambio en la ubicación seleccionada. Este evento, ***place_changed***, sucederá cuando el usuario haya seleccionado una opción de la lista de sugerencias (si fuera una página web normal también se podría ejecutar si el usuario pulsara *enter* estando escribiendo en la caja de texto).

En la función *placeChangedListener*, pues, se realiza una comprobación de que el objeto *Autocomplete* haya obtenido correctamente las coordenadas de la nueva ubicación, además de otra serie de comprobaciones y acciones necesarias para el correcto funcionamiento del filtro. Las coordenadas se obtienen de la siguiente manera:

```
var lat = autocompleteObj.getPlace().geometry.location.lat();
var lng = autocompleteObj.getPlace().geometry.location.lng();
```

Figura 115: Extracto de obtención de coordenadas del objeto Autocomplete de G.Maps

Por otro lado, en la pantalla del mapa se producía un problema con el mapa a la hora de navegar a otra pantalla como la de creación de un nuevo POI y se volvía al mapa. Al redirigir al usuario al mapa éste se **cargaba en ocasiones sólo parcialmente**, mostrando secciones grises en el mismo. Para solucionarlo se ha definido lo siguiente, que corrige este comportamiento cuando detecta que la vista que se va a mostrar es la del mapa.

```
$rootScope.$on('$stateChangeSuccess', function(event, toState, toParams,
                                                fromState, fromParams){
    if(toState.name == 'app.map'){
        $timeout(function() {
            Map.closeAndRemoveInfoWindow();
            var center = Map.getMap().getCenter();
            google.maps.event.trigger(Map.getMap(), 'resize');
            Map.getMap().setCenter(center);
        }, 1850);

    }
});
```

Figura 116: Extracto para evitar cargas parciales del mapa

Otro de los usos de la API de Google Maps se ha producido en el **filtro por ubicación**. Partiendo de una serie de POIs, se aplica sobre ellos un filtro en el cual se utiliza el método *bounds.contains(LatLng coords)*, el cual determina si una coordenadas están o no dentro de un área. La definición de dicha área se puede ver en la siguiente Figura 117, en la que se generan dos coordenadas, *sw* y *nw* (suroeste y noreste respectivamente) en base a una coordenada principal en cuyos valores se suma y resta una pequeña cantidad generando en la variable *bounds* un área de alrededor de 500m con centro la coordenada sobre la que se aplica el filtro de localización.

```
// override existing POIs by filtered ones (datasetResults.rows has the reference to real array)
var sw = new google.maps.LatLng(Number(lat) - 0.0035, Number(lng) - 0.0035);
var ne = new google.maps.LatLng(Number(lat) + 0.0035, Number(lng) + 0.0035);
var bounds = new google.maps.LatLngBounds(sw, ne); // set bounds (more or less 500m of radius)

var previousPOICount = datasetResults.rows.length;

// remove from the corresponding POIs array that ones that are not near the selected location
datasetResults.rows = datasetResults.rows.filter(poiIsContained, bounds);
```

Figura 117: Extracto de aplicación del filtro de localización

Una vez definida el área se aplica un filtro en base a ella. La función *poiIsContained* determina si cada POI del array *datasetResults.rows* está contenido o no en dicha área, obteniendo un nuevo array con todos los POIs que no cumplen la condición quitados del mismo. La implementación del filtro que se aplica en cada elemento del array se muestra en la siguiente Figura 117.

```
// function to be run for each element in 'filter' 'datasetResults.rows' array of applyLocationFilter(...)
// 'this' value is the optional parameter passed by 'filter' function: google maps' 'bounds' function
function poiIsContained(item, index, array) {
    var coordinatesLatLng =  item.latitudelongitude.split(",");
    var poiLatLng = new google.maps.LatLng(Number(coordinatesLatLng[0]), Number(coordinatesLatLng[1]));
    if(coordinatesLatLng[0] == null || coordinatesLatLng[1] == null){
        return false;
    }
    else{
        return this.contains(poiLatLng);
    }
}
```

Figura 118: Uso de Google Maps para el filtro de localización

Como se puede ver, la función *poiIsContained(...)* referencia el área *bounds* mediante *this*, de tipo *LatLngBounds*, y hace uso de su método *contains(...)* para determinar si las coordenadas del POI están contenidas o no en ella. Previamente se realiza una comprobación de que efectivamente el POI tiene especificadas unas coordenadas.

En resumen, se han mostrado algunos aspectos en el uso de la API de Google Maps para mostrar un mapa (y dibujar sobre él marcadores), implementar la lista de sugerencias de Google Places y aplicar un filtro de localización. La lógica e implementación subyacente para el control del mapa y filtro de POIs es mucho más compleja, tan sólo se han mostrado unos ejemplos concreto del uso de la API.

Filtrado de POIs

Por último se debe comentar a grandes rasgos la implementación del propio filtrado de POIs que ha sido extremadamente complejo programáticamente hablando. Es tal la cantidad de funciones implementadas para el CU de filtrar POIs que en el presente documento tan sólo se mencionan los aspectos que han debido tenerse en cuenta en su realización, si bien es cierto que se mostrará algún extracto de código quizás más relevante.

En líneas generales, se han tenido que tener en cuenta los siguientes aspectos:

- Los POIs se obtienen mediante consultas SQL sobre un *dataset* del ODS. Por tanto, en dicha llamada se realiza el filtro de una categoría y, si es necesario, por texto (con cláusulas WHERE).
- Los **POIs oficiales** se encuentran en **distintos *datasets***. Los de **ciudadanos**, en cambio, están todos en un **mismo *dataset*** del ODS.
- El uso del ODS para crear un nuevo POI requiere de la **autenticación** del usuario, para lo cual se debe indicar el *token* del usuario (que no haya expirado) a la hora de acceder a su API.
- La **configuración de categorías** debe realizarse de manera **externa y genérica**, de manera que tan sólo modificando dicho fichero de configuración se puedan añadir o quitar categorías. Esto incluye la necesidad de especificar:
    - El *id* del *dataset* de cada categoría oficial (además del *id* del *resource* del *dataset* al que se accede).
    - El *id* del *dataset* de POIs de ciudadanos (además del *id* del *resource* de dicho *dataset*) en el que se incluyen los de todas las categorías.
    - El icono a mostrar por cada categoría para un POI oficial.
    - El icono a mostrar por cada categoría para un POI de ciudadanos.
    - La opción de añadir una categoría que no existe de manera oficial, sólo se puede usar para POIs de ciudadanos.
    - La etiqueta a mostrar para cada categoría en los idiomas disponibles, a fin de mostrar la traducción correcta en el filtro de categorías.
- Se debe **gestionar la interfaz del propio filtro** (los distintos *checkbox*, la obtención de coordenadas mediante Google Places (y guardarlas mientras no se cambie esa ubicación) o a través del GPS, despliegue de las distintas secciones del filtro, etc.
- Se debe gestionar el **proceso de filtrado** (por cada categoría activada obtener los POIs oficiales del ODS (con el filtro de texto si es necesario) y

aplicar después el filtro de localización, para repetir el proceso con los POIs de ciudadanos. Los filtros de texto, localización y ver también de ciudadanos sólo se pueden activar si al menos hay una categoría seleccionada, ya que se aplican sobre cada una de ellas.

Además, se debe reaplicar todo el filtro sólo cuando haga falta. Si no hace falta, tratar de **minimizar el coste del proceso** aplicando sobre los POIs previamente filtrados el nuevo filtro especificado, en vez de aplicar todo el proceso. A saber:

- **Activar categoría o texto.** El filtro de categoría y/o de texto requiere reaplicar todos los filtros restantes (localización y ver de ciudadanos), ya que se obtienen nuevos POIs del ODS.
- **Desactivar categoría.** Tan sólo elimina los POIs guardados de esa categoría y, si está el filtro de ver de ciudadanos activado, también quita los POIs de ciudadanos de esa categoría.
- **Activar localización (diferenciar si se cambia GPS por Google Places y viceversa, o si se activa una de ellas no estando ninguna activada previamente).** El filtro de localización requiere reaplicar todos los filtros para cada categoría seleccionada sólo cuando previamente se había aplicado otro filtro de localización. Si se cambia el filtro de localización de uno a otro hace falta repetir todo el proceso, sino sólo se debe aplicar el filtro de localización sobre los ya encontrados.

  Es decir, si anteriormente están filtrados por ejemplo por GPS, esto significa que hay POIs que se han eliminado de los obtenidos con los demás filtros. Por tanto, en ese caso al cambiar el filtro de ubicación GPS al filtro de ubicación seleccionada con Google Places hace falta reaplicar primero los demás filtros para obtener todos esos POIs que se habían eliminado por la ubicación GPS, y eliminar después los que no cumplan la condición de la nueva ubicación.
- **Desactivar localización o texto.** Al quitar el filtro de localización o de texto, se debe procesar todo el filtrado de nuevo (porque se deben obtener todos los POIs de nuevo sin esos parámetros).
- **Activar ver de ciudadanos también.** Al activar el filtro de ver de ciudadanos sólo se debe aplicar todo el proceso del filtrado (obtener del ODS, si es necesario en base a texto también, y terminar aplicando el filtro si es necesario) para cada categoría, sin modificar el resto de POIs filtrados.

    - **Desactivar ver de ciudadanos también.** Tan sólo elimina los POIs de ciudadanos almacenados para cada categoría seleccionada.

- Dada la naturaleza del proceso de filtrado, en la cual se realizan llamadas al ODS o se pueden obtener las coordenadas GPS, hay que tener mucho cuidado con el flujo del proceso. Es decir, la lógica del proceso funciona sólo si se realiza una **ejecución secuencial** del mismo, por tanto se deben controlar todas las llamadas asíncronas y que varían el tiempo de ejecución, haciendo que el proceso secuencial del filtrado avance cuando terminen estas llamadas asíncronas.
- Se deben gestionar los **marcadores a visualizar en el mapa** una vez que se ha terminado el proceso de filtrado de POIs, incluyendo el icono a mostrar, los *infoWindow* o pequeños paneles que se muestran al pulsar en cada uno de ellos, o el *viewport* (área de visualización del mapa en cuando a zoom, etc.) para mostrar el mapa en función de los POIs obtenidos (alejar la vista, acercarla, mostrar el icono de la localización indicada para dicho filtro y mostrar los POIs encontrados a su alrededor, etc.).
- Se deben **gestionar errores** en el acceso al ODS, obtención de coordenadas (GPS o de Google Places) y demás.

Como se puede apreciar, son muchos los aspectos a tener en cuenta en el proceso de filtrado de POIs y visualización de los mismos en el mapa. Para la implementación ha resultado absolutamente imprescindible un diseño software coherente y separado en distintas capas, y también la realización de funciones genéricas para poder reutilizar y ejecutar distintos tipos de filtros en función de la parte del proceso de filtrado que se debe ejecuta, alguna o toda.

Dicho esto, se han creado dos servicios para repartir algunas responsabilidades. El primero, *'Map',* tal y como se ha dicho anteriormente, se encarga de la gestión del mapa, marcadores, el objeto *Autocomplete* de Google Maps y demás. En el segundo, *'FilteredPOIs'*, se almacenan los POIs oficiales y de ciudadanos obtenidos para cada categoría, y ofrece funciones para reaplicar los filtros por separado o para obtener los POIs filtrados.

En cuanto al **control de la ejecución secuencial** de diferentes funciones que pueden tener una ejecución asíncrona, se diferencias **dos casos**. El primero simplemente es el control de la ejecución de una función asíncrona en el sentido de detectar cuándo ha terminado para proseguir después con otras funciones. Esto se ha conseguido gracias al uso exhaustivo del elemento ***$q*** de AngularJS, un *promise* que es el devuelto por la función asíncrona para que en el sitio donde se la llama se pueda configurar qué se debe ejecutar al término de su ejecución. En las siguientes Figuras 119 y 120 se muestra un ejemplo de esto.

```
// función con ejecución asíncrona
function funcionAsincrona(){
    var promise;
    promise = $q(function (resolve, reject) {

        ... // ejecución asíncrona

        if(success){
            resolve('ok!');
        }else{
            reject('error!');
        }
    });

    return promise;
}
```

Figura 119: Extracto ejemplo de uso de promise's en funciones asíncronas

```
// llamada a función asíncrona
functionAsincrona()
    .then(successCallback, errorCallback);

// se lanza cuando la función asíncrona haya terminado satisfactoriamente
function successCallback(msg){
    console.log('Mensaje de respuesta de la función asíncrona: ', msg); // output 'ok!'
    ...
};

// se lanza cuando la función asíncrona haya terminado con error
function errorCallback(msg){
    console.log('Mensaje de respuesta errónea de la función asíncrona: ', msg); // output 'error!'
    ...
};
```

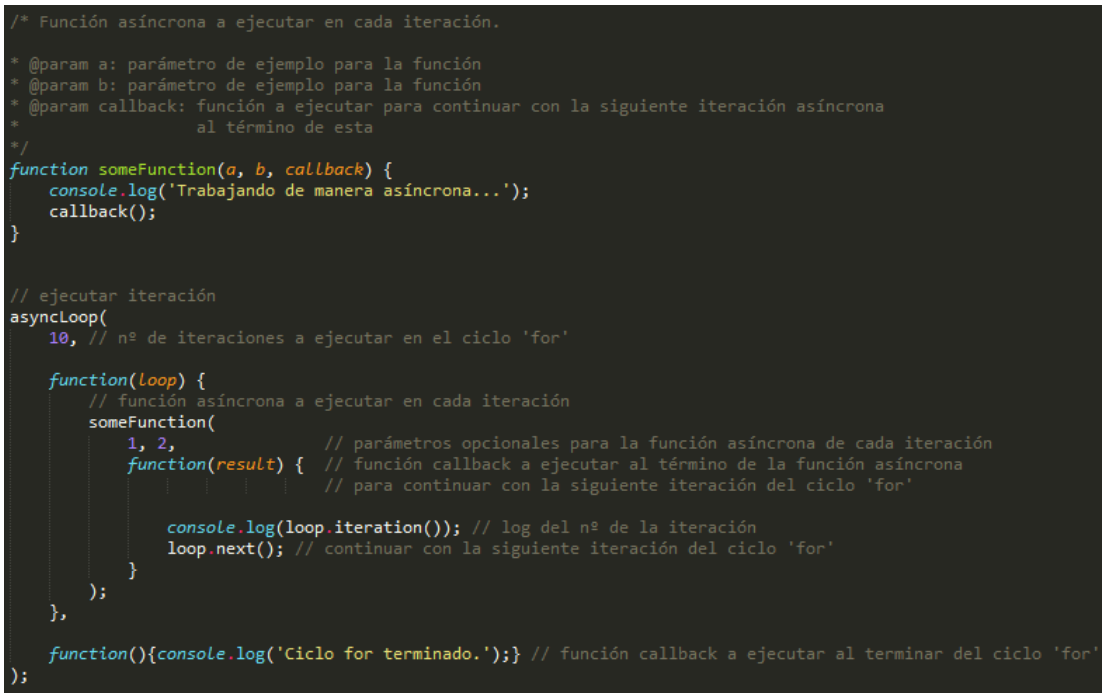
Figura 120: Extracto ejemplo de uso de promise para forzar ejecución secuencial

En la Figura 120, además del bloque *.then(success, error*) en el que se definen las funciones a ejecutar en función de las respuestas del *promise*, se debe comentar que también se podría añadir un bloque ***finally*** para ejecutar una función siempre que termine la ejecución asíncrona de la función, independientemente de si lo ha hecho satisfactoriamente o no. En BilbOn el uso de este tipo de elementos de AngularJS ha sido básico para resolver este problema de ejecuciones asíncronas y secuenciales, aunque la implementación realizada es mucho más compleja que en los ejemplos básicos mostrados. Los mensajes que se pueden indicar en el *resolve()* y *reject()* se han utilizado para devolver respuestas o gestionar posibles errores.

El segundo caso de control de ejecuciones asíncronas se ha producido por la necesidad de repetir el **proceso de filtrado** de POIs por separado para **cada categoría, de forma iterativa**. Para forzar una ejecución secuencial de un bucle iterativo que repite un proceso de filtrado asíncrono para cada categoría activada, la técnica aplicada ha sido la mostrada, de manera simplificada, en los siguientes extractos.

```
/* Función para gestionar el ciclo 'for' de iteraciones asíncronas.

* @param iterations: nº de iteraciones a ejecutar en el ciclo 'for'
* @param func: función asíncrona a ejecutar en cada iteración
* @param callback: función a ejecutar al término del ciclo 'for'
*/
function asyncLoop(iterations, func, callback) {
    var index = 0;
    var done = false;
    var loop = {
        next: function() {
            if (done) {
                return;
            }

            if (index < iterations) {
                index++;
                func(loop);

            } else {
                done = true;
                callback();
            }
        },

        iteration: function() {
            return index - 1;
        },

        break: function() {
            done = true;
            callback();
        }
    };
    loop.next();
    return loop;
}
```

Figura 121: Extracto de ejemplo de función para ciclo 'for' asíncrono



```
/* Función asíncrona a ejecutar en cada iteración.

* @param a: parámetro de ejemplo para la función
* @param b: parámetro de ejemplo para la función
* @param callback: función a ejecutar para continuar con la siguiente iteración asíncrona
*                  al término de esta
*/
function someFunction(a, b, callback) {
    console.log('Trabajando de manera asíncrona...');
    callback();
}


// ejecutar iteración
asyncLoop(
    10, // nº de iteraciones a ejecutar en el ciclo 'for'

    function(loop) {
        // función asíncrona a ejecutar en cada iteración
        someFunction(
            1, 2,               // parámetros opcionales para la función asíncrona de cada iteración
            function(result) {  // función callback a ejecutar al término de la función asíncrona
                                // para continuar con la siguiente iteración del ciclo 'for'

                console.log(loop.iteration()); // log del nº de la iteración
                loop.next(); // continuar con la siguiente iteración del ciclo 'for'
            }
        );
    },

    function(){console.log('Ciclo for terminado.');} // función callback a ejecutar al terminar del ciclo 'for'
);
```

Figura 122: Extracto de ejemplo de uso de ciclo 'for' asíncrono

En las Figuras 121 y 122 se comenta para qué sirve cada función y parámetro para poder implementar la ejecución de un ciclo '*for*' con ejecuciones asíncronas para las iteraciones. En la implementación del filtrado, como se ha dicho, esto ha servido para iterar por las categorías activadas, y ejecutar para cada una la función asíncrona correspondiente para aplicar el filtro, primero de POIs oficiales y después de POIs de ciudadanos si estuviera aplicado dicho filtro.

Para ello, en lo que sería la función equivalente a *someFunction(...)* de la Figura 122 (la función asíncrona a ejecutar), el *callback()* no se ejecuta hasta haber terminado la ejecución asíncrona de la función de filtrado correspondiente. Esto se ha hecho con la técnica mencionada en el primer punto, el uso de ***promise's*** (ver la Figura 123).

```
// function to run in every loop of the async cycle to apply enabled filters for all selected categories
// see 'asyncLoopForCategoriesFilter(...)', used for asynchronous 'for' cycle
function filterCategoryLoopFunction(categoryCustomNumericId, categoryTranslatedName, iterationIndex, callback) {
    console.log('Starting iteration ', iterationIndex);
    // get category official POIs and filter by text and location if neccesary; repeat for citizens' POIs
    // ensure that the functions have been finished before calling the callback function for reload the markers
    getPOIsWithAllFilters(categoryCustomNumericId, categoryTranslatedName, true)
    .then(function(){
        getPOIsWithAllFilters(categoryCustomNumericId, categoryTranslatedName, false)
        .then(function(){
            callback();
        });
      });
};

// function to run in every loop of the async cycle to apply enabled filters for all selected categories
// see 'asyncLoopForCategoriesFilter(...)' in '$scope.callAlsoCitizensFilter', used for asynchronous 'for' cycle
function filterCategoryCitizenLoopFunction(categoryCustomNumericId, iterationIndex, callback) {
    console.log('Starting iteration ', iterationIndex);
    // get category citizens' POIs and filter by text and location if neccesary
    getPOIsWithAllFilters(categoryCustomNumericId, null, false)
    .then(function(){
        callback();
    });
};
```

Figura 123: Implementaciones del 'someFunction(...) de la Figura 122 para el filtrado

En esta Figura 123 se pueden ver las funciones asíncronas que se ejecutan en cada iteración del ciclo que analiza las categorías activadas, cuando se activa una categoría en la función superior de la imagen, y cuando se activa el filtro de ver de ciudadanos en la parte inferior. Se muestra claramente, al **activar una categoría**, que **aplica todos los filtros** activados para esa categoría (internamente obtendrá POIs de esa categoría, y si hiciera falta también filtrado por texto y por ubicación si es necesario) para POIs **oficiales**, y **repite** el proceso para los de **ciudadanos**, antes de seguir con la **siguiente iteración, es decir, con la siguiente categoría activada**. La función de la parte interior sigue la misma estrategia, pero en este caso sólo aplica todos los filtros para cada categoría para POIs de ciudadanos.

La implementación realizada se ha basado en esta técnica (aunque ha sido necesario modificarla de acuerdo a las necesidades del filtrado y se han tenido que tener en cuenta otros aspectos), obtenida de un foro de la comunidad de desarrolladores *StackOverflow:* http://stackoverflow.com/questions/4288759/asynchronous-for-cycle-in-javascript.

6.4.3.1. Problemas encontrados

Los principales problemas encontrados son los que se han ido destacando. Sobre todo la gestión del filtro de POIs y todo lo que ello conlleva (ejecuciones asíncronas, control de todo lo relacionado con el mapa y del objeto *Autocomplete* de Google Maps,...). Si bien el desarrollo y estructuración de la implementación de todo esto ha sido muy compleja, bien es cierto también que ha implicado un aprendizaje en el uso de muchas técnicas que no se conocían.

A todo esto hay que añadir lo ya comentado para la implementación de Bilbozkatu. El propio estudio e integración de los *frameworks*, tecnologías, herramientas, librerías y técnicas usadas, prácticamente empezando de cero en todo ello, ha supuesto un problema aunque mejor entendido como desafío. Todo esto además del uso de los componentes de WeLive, la mayoría reutilizado de la app Bilbozkatu en este caso.

6.4.4. **Verificación y pruebas**

En este apartado se aplica todo lo explicado en el mismo apartado de la app Bilbozkatu, en el punto 6.3.4. El desarrollo de la app se ha ido realizando con pruebas constantes gracias a los "cambios en caliente" que el servidor de Ionic proporciona y a la consola de desarrolladores del navegador Chrome en el que se ha ido testeando.

Al igual que en Bilbozkatu, hacia el final del desarrollo de la app, personal de Eurohelp y de Deusto externo al desarrollo de la app ha realizado numerosas pruebas y test en la app. Afortunadamente, los únicos *feedbacks* que estas personas han dado han sido sobre mejoras, y no *bugs*. Al no consistir en mejoras importantes, tan sólo eran algunos detalles que otros, se ha ido integrando en la app.

6.4.5. **Publicación**

Los pasos para la generación del fichero .apk correspondiente a la plataforma Android y posterior publicación en Google Play son los mismos que los indicados en el apartado de la app Bilbozkatu (ver el punto 6.3.5).

En la siguiente Figura 124 aparecen los permisos que la aplicación requiere en cuanto al acceso a características nativas de Android. Es necesario permitir el acceso a la ubicación GPS del dispositivo para BilbOn (gracias al plugin de Cordova de geolocalización), para poder instalarla.

En la ejecución de la app, la "ubicación" debe estar activada en el dispositivo para poder obtener la localización vía GPS o redes wifi. Si no lo está, al fallar la obtención de coordenadas la app informa al usuario de que no se han podido obtener, y sugiere que revise si esta característica está activada en el dispositivo. Si no se hace sólo se podría filtrar utilizando Google Places, como se ha dicho.

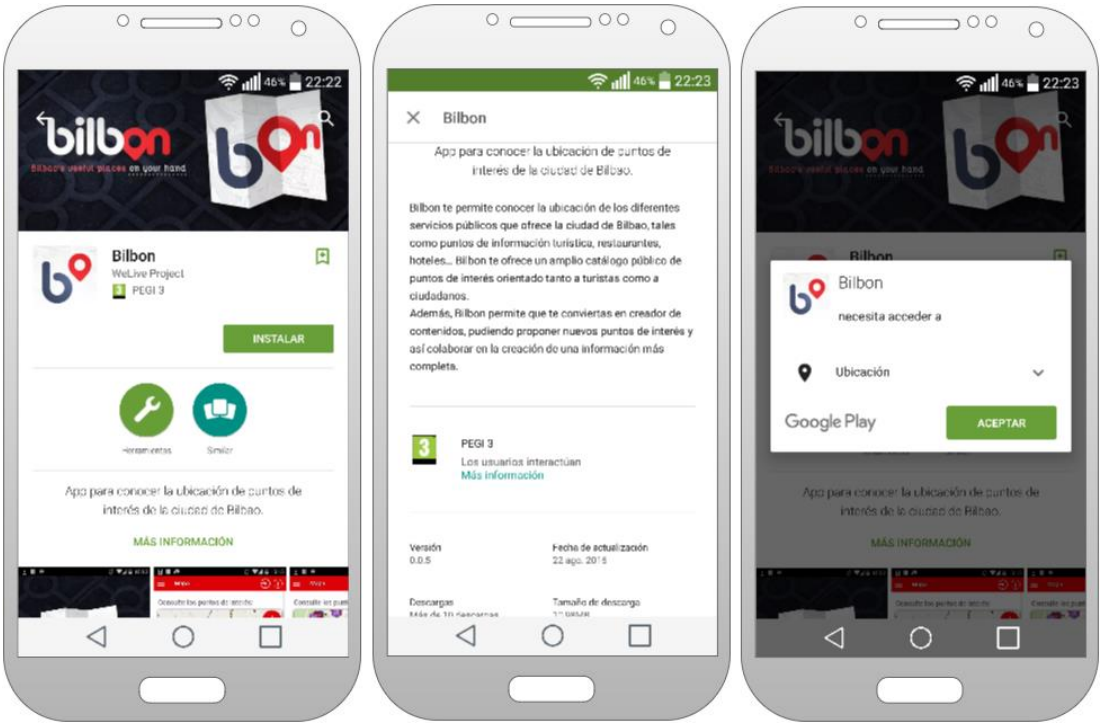


Figura 124: BilbOn en Google Play y permiso de instalación

Capítulo 7

7. GESTIÓN DEL PROYECTO

En este capítulo 7 se reúnen los aspectos correspondientes a la gestión del proyecto. Se incluyen los apartados de gestión del alcance, estudio de la viabilidad, gestión de cambios, análisis de costes (con la comparativa estimación vs real) y planificación.

El alcance y dedicación del proyecto se ha visto alterado a lo largo del desarrollo del mismo debido a nuevas exigencias del cliente (responsables del proyecto WeLive en este caso), y a los cambios que se han ido realizando ocasionalmente en la especificación de uso de los componentes de la plataforma WeLive, cosa que ha tenido un impacto al integrarlos en las aplicaciones móviles desarrolladas.

7.1. Gestión del alcance

Tal y como se ha comentado en el capítulo 3 del presente proyecto referente al alcance del proyecto, éste ha sufrido variaciones a lo largo del desarrollo del mismo debido a nuevas exigencias del cliente (responsables de WeLive) que evidentemente no se identificaron al inicio. Por otro lado, en relación al uso de componentes de WeLive, a pesar de que inicialmente se sabía que se iban a tener que integrar, debido a que su desarrollo se ha realizado en paralelo al presente proyecto se han producido en ocasiones cambios en la especificación de uso que también han supuesto un impacto. A continuación se enumeran, en líneas generales, los principales puntos que han supuesto estos cambios.

Las principales características inicialmente ignoradas que se han tenido que añadir son:

- Integración del componente de WeLive *LoggingBB* en las apps para el registro en él de una serie de logs o KPIs (acciones específicas del usuario en la app) definidos para cada una. Posteriormente, securización en el uso del componente, cosa que ha provocado tener que adecuar el uso de dicho componente.
- Integración del cuestionario de opinión del usuario en las apps.
- Integración del *splash screen* realizado por Alex Novoa, diseñador gráfico de Eurohelp.
- Integración de la pantalla de Términos y Condiciones de Uso al inicio de la app en base al texto definido por una comisión del proyeto WeLive, de acuerdo a la legislación actual (para la LOPD, etc.).
- En los *Building Blocks* implementados (*UsersFeedbackBB* y *BilbozkatuBB*), generación automática del WADL con el que ofrecer la especificación de uso de los mismos en formato XML.
- En los BBs añadir soporte para PostgreSQL para el acceso a datos (inicialmente era para MySQL), con la subyaciente formación en el uso y la definición del modelo de datos para que sea equivalente al de MySQL.

En segundo lugar, los puntos que inicialmente se identificaron pero se ignoraba su coste (desconocimiento del componente o plataforma, cambios en la especificación de uso...) son:

- Despliegue de los *Building* Blocks en *Cloud Foundry*. Necesaria la formación básica para el uso de dicha plataforma y configuración de los BBs para su despliegue. Problemas en la configuración de la cuenta *Cloud Foundry* utilizada para desplegar los servicios, gestionada por la empresa italiana FBK, han supuesto una dedicación innecesaria al tratarse finalmente de un problema de FBK.

- Integración del componente AAC BB de WeLive en las apps e implementación del proceso de autenticación definido con OAuth 2.0.
- Uso e integración del componente ODS (*Open Data Stack*) en la app BilbOn. Posteriormente, cambios en la especificación del mismo, además de la securización en operaciones de creación de POIs.

Cabe añadir que el testeo interno hecho por personal de Eurohelp y Deusto ajeno al desarrollo ha supuesto también el tener que realizar ciertas mejoras en las apps. Si bien no han tenido mucha dedicación al tratarse más de detalles o mejoras de usabilidad, el hecho de que se trate de una app móvil provoca siempre que haya detalles que pulir o mejorar.

7.2. Estudio de la viabilidad

En esta sección se muestra el estudio de la viabilidad realizado para el presente proyecto de fin de grado. Se mencionan los riesgos que podrían provocar el incumplimiento de los objetivos marcados e, incluso, la insatisfacción del cliente, en este caso los responsables del proyecto WeLive en general y los de dicho proyecto en la empresa Eurohelp en concreto. Se añade también el plan de contingencia definido a grandes rasgos.

7.2.1. **DAFO**

Este es el análisis DAFO que muestra las fortalezas, debilidades, oportunidades y amenazas del proyecto a nivel general.

	Aspectos positivos	**Aspectos negativos**
Origen interno	**FORTALEZAS** • Proyecto motivador • Experiencia previa en la empresa • Formación en múltiples tecnologías, *frameworks*, técnicas y herramientas	**DEBILIDADES** • Ningún conocimiento de los frameworks y técnicas a utilizar • Muchos artefactos software a desarrollar
Origen externo	**OPORTUNIDADES** • Experiencia laboral y a nivel europeo • Auge de los desarrollos híbridos para apps • Proyecto multidisciplinar	**AMENAZAS** • Documentación obsoleta de componentes de la plataforma WeLive • Poco soporte a proyectos de i+D+i

Tabla 14: Análisis DAFO del proyecto

Los aspectos identificados en el análisis DAFO de la tabla 14 son los que se han ido mencionando y destacando a lo largo del presente documento. Si bien es cierto que se partía del desconocimiento en cuanto a tecnologías, *frameworks*, etc. a utilizar, a pesar de que el camino no haya sido fácil la formación y aprendizaje en los meses que ha durado el proyecto ha sido muy potente, además de la propia experiencia de trabajar en un entorno laboral en el ámbito de un proyecto a nivel europeo.

7.2.2. **Identificación de los riesgos**

A continuación se enumeran los riesgos identificados más relevantes que pueden afectar negativamente el transcurso normal del proyecto.

- Estimaciones no realistas. Normalmente las estimaciones y las dedicaciones no suelen concordar al 100%, pero al no tener mucha experiencia de trabajo en proyectos grandes (además europeo en este caso), unido a la realización del mismo en empresa con la colaboración de otras, y con la necesidad de formarse desde cero en numerosos *frameworks*, conceptos, técnicas y herramientas, hace que las desviaciones puedan aumentar y el tiempo de desarrollo alargarse.
- Disponibilidad de colaboradores. El hecho de tener que integrar componentes de la plataforma WeLive que están en desarrollo por parte de otras empresas de WeLive, y con cambios constantes, hace necesario que dichos desarrolladores estén en la medida de lo posible disponibles para poder contactar en caso de tener problemas.
- Variación del alcance. Al tratare de un proyecto de i+D con apps piloto que deben ir pasando fases e ir mejorando, es factible que el alcance sufra modificaciones.
- Librerías con comportamiento no esperado. Además de los propios componentes de WeLive mencionados, las apps hacen uso de librerías (algunas de las cuales descubiertas en el transcurso del proyecto) que es posible que no ofrezcan el comportamiento esperado.
- Pérdida de información. Borrados accidentales, equipos averiados o cualquier otra cosa pueden hacer que se pierda parte del trabajo realizado.
- Problemas de salud. Siempre es posible que sucedan imprevistos relacionados con la salud que impidan el trancurso normal del proyecto.

7.2.3. **Cuantificación de los riesgos y plan de contingencia**

En la siguiente tabla se analizan los riesgos identificados en el punto anterior, cuantificándolos e indicando las acciones preventivas, mitigadoras o correctivas correspondientes.

	Categoría	Probabilidad / Impacto	Acción preventiva	Acción mitigadora	Acción correctiva
Estimaciones no realistas	Planif.	Prob: Alta Imp: Alto	Consultar con profesionales más experimentados.	Realizar replanificaciones periódicas.	-
No disponibilidad de colaboradores	Técnico Ejecución	Prob: Media Imp: Bajo	Disponer de la documentación necesaria para el uso de los componentes externos.	Replanificar la integración del componente.	-
Variación en el alcance	Planif.	Prob: Media Imp: Alto	Consultar con los responsables de WeLive futuras características, aunque probablemente no se disponga de esa información.	Desarrollo modular, de manera que cambios en un módulo no afecten al resto. Aplicar buenas prácticas del desarrollo del software.	-
Comportamiento librerías inesperado	Ejecución	Prob: Baja Imp: Medio	Se harán testeos previos a la integración de cualquier tecnología externa.	Se sustiuirá por otra librería o desarrollo a medida.	Se estudiarán las alternativas y se seleccionará la más adecuada.
Pérdida de información	Sist. info.	Prob: Baja Imp: Alto	Realizar copias de seguridad periódicas y tener el proyecto sincronizado en Dropbox constantemente.	Recuperar una copia de seguridad anterior u obtenerlo de Dropbox si es posible. Sino, utilizar herramientas especializadas en la recuperación de archivos.	Especificar las copias de seguridad diarias, de manera que la restauración de archivos sea completa.
Problemas de salud	Personal	Prob: Baja Imp: Medio	Aumentar las medidas de seguridad y protección pertinentes.	Continuar desarrollo desde casa en la medida de lo posible, y replanificar.	-

Tabla 15: Cuantificación de riesgos y plan de contingencia

Cabe añadir que un punto importante en el plan de contingencia es también el aplazar la fecha de entrega del proyecto a una convocatoria posterior. Así, si inicialmente se había pensado en desarrollar el proyecto hasta septiembre, finalmente se ha retrasado a la siguiente convocatoria de septiembre.

7.3. Gestión de cambios

Las alteraciones en el alcance del proyecto han sido una constante, tanto por la naturaleza del mismo (proyecto de i+D con aplicaciones piloto las cuales pasan revisiones por parte de la Comisión Europea y sufren una constante evolución) como por tratarse de apps móviles que, como es normal, siempre hay detalles que pulir o mejorar.

Sabiendo esto se ha asumido que el desarrollo podía sufrir retrasos. Como se acaba de comentar, aunque la intención inicial era la de terminar el desarrollo del proyecto para julio, estos cambios han provocado su retraso hasta la siguiente convocatoria de septiembre. Es cierto que quizás era posible disminuir el alcance del proyecto de fin de grado y mantener el proyecto principal en la empresa aparte, pero la la posibilidad de retrasar sin ninguna consecuencia más que la dedicación realizada a la convocatoria de septiembre, se ha decidido seguir adelante con los cambios en el proyecto de fin de grado.

7.4. Análisis de costes

En esta sección se muestran los costes en horas de las tareas principales realizadas a lo largo del proyecto, algunas de las cuales analizadas en el capítulo 6 del presente documento. Estas corresponden tanto a la formación y desarrollo como a la gestión del proyecto o elaboración de la memoria.

En cuanto al desarrollo se especifican cada uno de los cuatro componentes software realizados: los servicios web *UsersFeedbackBB* y *BilbozkatuBB* y las apps *Bilbozkatu* (que utiliza los servicios web mencionados) y *BilbOn.* Algunas de las características implementadas en un artefacto software han podido ser reutilizadas en otro disminuyendo el coste de implementación, aunque cada artefacto tiene sus peculiaridades.

Evidentemente también se deben indicar las tareas de formación más importantes realizadas, que en este caso consisten sobre todo en el estudio de los *frameworks* y herramientas que proporcionan Ionic, AngularJS y Apache Cordova (en cuanto a Cloud Foundry se computa en la fase de despliegue). Por último se indican las tareas propias de la gestión del proyecto y elaboración del trabajo académico, a saber, la presente memoria y la preparación para la defensa del proyecto de fin de grado (aunque este último aún no se ha terminado).

Actividad	Tarea	Estimación	Dedicación	Desviación
Inicio	Inicio y documentación proyecto WeLive	10h	14h	+4h (40%)
Formación en tecnologías y herramientas	Estudio Spring Boot	16h	13h	-3h (-18,75%)
	Estudio Ionic (*framework*, etc.)	20h	23h	+3h (15%)
	Estudio AngularJS	16h	15h	-1h (-6,25%)
	Estudio Apache Cordova	12h	3h	-9h (-75%)
	PostgreSQL y cliente	12h	7h	-5h (-41,67%)
Desarrollo *UsersFeedback BB*	Análisis	3h	2h	-1h (-33,33%)
	Diseño	10h	8h	-2h (-20%)
	Implementación	20h	30h	+10h (50%)
	Verificación y pruebas	14h	10h	-4h (-28,57%)
	Despliegue	4h	15h	+11h (+275%)
Desarrollo *BilbozkatuBB*	Análisis	2h	2h	0h (0%)
	Diseño	2h	2h	0h (0%)
	Implementación	12h	10h	-2h (-16,67%)

	Verificación y pruebas	3h	3h	0h (0%)
	Despliegue	2h	1h	-1h (-50%)
Desarrollo Bilbozkatu app	Análisis	8h	10h	+2h (25%)
	Diseño	25h	34h	+9h (36%)
	Implementación	80h	119h	+39h (48,75%)
	Verificación y pruebas	2h	2h	0h (0%)
	Despliegue	3h	2h	-1h (-33,33%)
Desarrollo BilbOn app	Análisis	10h	5h	-5h (-50%)
	Diseño	20h	18h	-2h (-10%)
	Implementación	60h	85h	+25h (41,67%)
	Verificación y pruebas	2h	2h	0h (0%)
	Despliegue	2h	1h	-1h (-50%)
Gestión del proyecto	Gestión del alcance	5h	4h	-1h (-20%)
	Estudio de la viabilidad	3h	2h	-1h (-33,33%)
	Reuniones de gestión y seguimiento en empresa	-	13h	-

Trabajo académico	Redacción de la memoria	80h	91h	+11h (13,75%)
	Preparación de la defensa	12h*	-	-
TOTAL		458h	546h	+78h (+19,21%)

Tabla 16: Costes y desviaciones de las tareas del proyecto

* En el total de la estimación no se han incluido las 12 horas estimadas para la preparación de la defensa.

Es importante señalar que no se ha hecho una planificación muy detallada al inicio del proyecto por el desconocimiento que se ha tenido a la hora de estimar muchas tareas. Esto es debido a que, como se ha podido ver a lo largo de los anteriores capítulos, los artefactos software desarrollados han requerido de mucho tiempo de formación y de integración de componentes externos, de WeLive y otros.

Por tanto, siguiendo la metodología analizada en el capítulo 4, los costes de las planificaciones a corto plazo están incluidos en las **reuniones de seguimiento y control** realizadas. En cada una de ellas se exponían los avances realizados y se definían y estimaban los siguientes pasos a grandes rasgos, de acuerdo a una idea general en cuanto a los hitos del proyecto.

Otra tarea como la definición del **alcance** tampoco se ha especificado en la anterior Tabla 16 debido a que también se ha hecho al inicio del proyecto en reuniones con el responsable de WeLive en la empresa Eurohelp. Además, el alcance general ha ido evolucionando a lo largo del ciclo de vida del proyecto.

Por último, se debe aclarar que las dedicaciones mostradas en relación a la formación en diferentes tecnologías y herramientas no son exactas. Si bien se han realizado esas tareas de formación previa para ir aprendiendo tanto los conceptos como el uso de esas tecnologías, es evidente que al comenzar la propia implementación ha sido necesario continuar investigando numerosas cosas. Es por ello que las dedicaciones de la actividad "Formación en tecnologías y herramientas" no suponen todo el tiempo dedicado a ello, el cual se integraría en la fase de implementación de los distintos artefactos software.

7.4.1. Estimación vs real

En la Tabla se puede apreciar que se ha producido una desviación del 19,21% en el tiempo dedicado con respecto al estimado. Es una desviación considerable pero lógica dada la magnitud del proyecto. La necesidad de formarse en todas las tecnologías y *frameworks* analizados en el presente documento, partiendo de

cero en todos ellos excepto en el uso de la API de Google Maps (con el que ya se tenía cierta experiencia previa en la propia empresa Eurohelp) y el uso de las tecnologías web más conocidas como HTML5, JavaScript y CSS, aunque con esta última no se había trabajado de manera tan exhaustiva, han provocado esta desviación. También ha tenido que ver la integración de componentes de WeLive en colaboración con otras empresas.

Por otro lado, se deben mencionar un par de apuntes con respecto a algunos costes mostrados en la tabla. En primer lugar, la desviación exagerada que ha sucedido en la fase de despliegue del servicio *UsersFeedbackBB*, del 275%, se ha producido por problemas en el uso de Cloud Foundry que se han intentado solucionar por todas las vías posibles pero que, al final, ha resultado ser un problema de configuración de la empresa LKS de WeLive, la cual es la encargada de proporcionar las cuentas de Cloud Foundry a los desarrolladores de WeLive. Hasta que se ha dado con la solución han pasado esas horas en las que se ha estado intentando realizar el despliegue del servicio.

Por otro lado, la escasa dedicación que se muestra en la tabla para las pruebas de las apps no ha sido exactamente así. Al final del desarrollo se han testeado las apps en diferentes dispositivos, pero a lo largo del desarrollo evidentemente se han ido probando constantemente las implementaciones realizadas. Además, tal y como se ha explicado en las secciones correspondientes a las verificaciones y pruebas de las apps, personal de Eurohelp y Deusto han trabajado una vez finalizadas las apps en una tarea de testing exhaustiva de las mismas, detectando sobre todo posibles pequeñas mejoras más que errores.

Se puede apreciar también en líneas generales que el desarrollo de *BilbozkatuBB* se ha producido más rápido que *UsersFeedbackBB*, ya que todos los diseños, técnicas e implementaciones de *UsersFeedbackBB* se han podido reutilizar. Algunos diseños, implementaciones (como el sistema de autenticación o cuestionario de opinión) de Bilbozkatu también se han reutilizado en la app BilbOn. Por supuesto, todo el conocimiento adquirido también ha supuesto de gran ayuda al desarrollar el segundo servicio o la segunda app.

7.5. Planificación

En este apartado se muestran los diferentes diagramas relacionados con la planificación como son el EDT o Estructura de Descomposición de Trabajo, el diagrama de Gantt y el diagrama de hitos. Dada la naturaleza del proyecto, al inicio del mismo no se realizó una planificación detallada de las tareas a realizar. De hecho, el responsable de WeLive y la directora del proyecto en la empresa Eurohelp sólo mencionaron comenzar con el desarrollo del *Building Block UsersFeedback* dentro del proyecto WeLive, y que más adelante probablemente se usara en alguna app que hubiera que realizar. Pero no se detalló nada al respecto, ni se definieron las apps a desarrollar.

Ya había definidas una serie de apps a hacer para la ciudad piloto de Bilbao. Una vez se había terminado *UsersFeedbackBB*, se decidió sobre la marcha aprovechar ese servicio en la app Bilbozkatu. Más adelante, tratando de aprovechar el conocimiento adquirido en cuando a Ionic en el desarrollo de apps híbridas, se decidió proseguir con BilbOn pensando que se podría hacer más rápido. No obstante, nuevas complejidades en el uso de un nuevo componente de WeLive (el ODS), el filtro de POIs y nuevas características a integrar en todas las apps de WeLive han hecho que el ciclo de vida del proyecto se alargue.

Dicho esto, evidentemente los diagramas que se muestran a continuación corresponden a la planificación realizada de manera general según se ha ido avanzando, y es por ello que no se muestra una planificación inicial para comparar con los plazos que finalmente se han producido.

Antes de mostrar el EDT cabe añadir que el desarrollo del proyecto se ha producido primero con un convenio de PFG entre la empresa Eurohelp, la Universidad del País Vasco y el autor del presente proyecto, segundo en un pequeño periodo fuera de la empresa pero manteniendo el contacto, y el tercero con un contrato de trabajo.

7.5.1. EDT

A continuación se muestra el EDT simplificado, ya que cada elemento de la sección "Producto" realmente incluye las tareas de análisis, diseño, implementación, verificación y pruebas y, finalmente, despliegue.

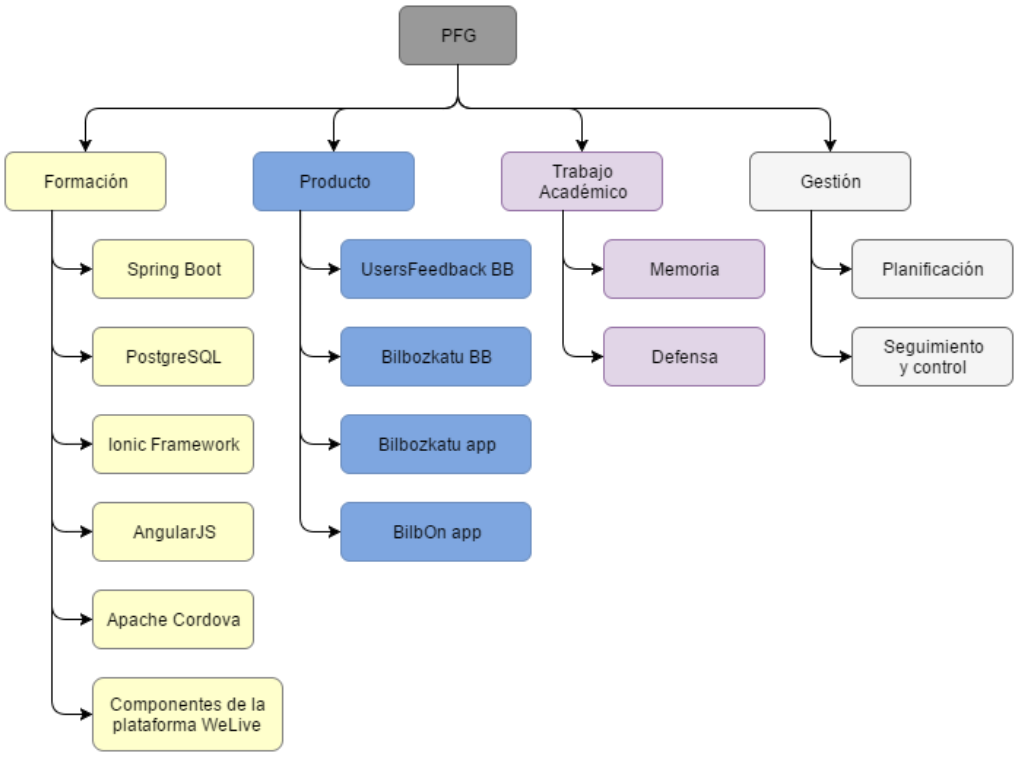
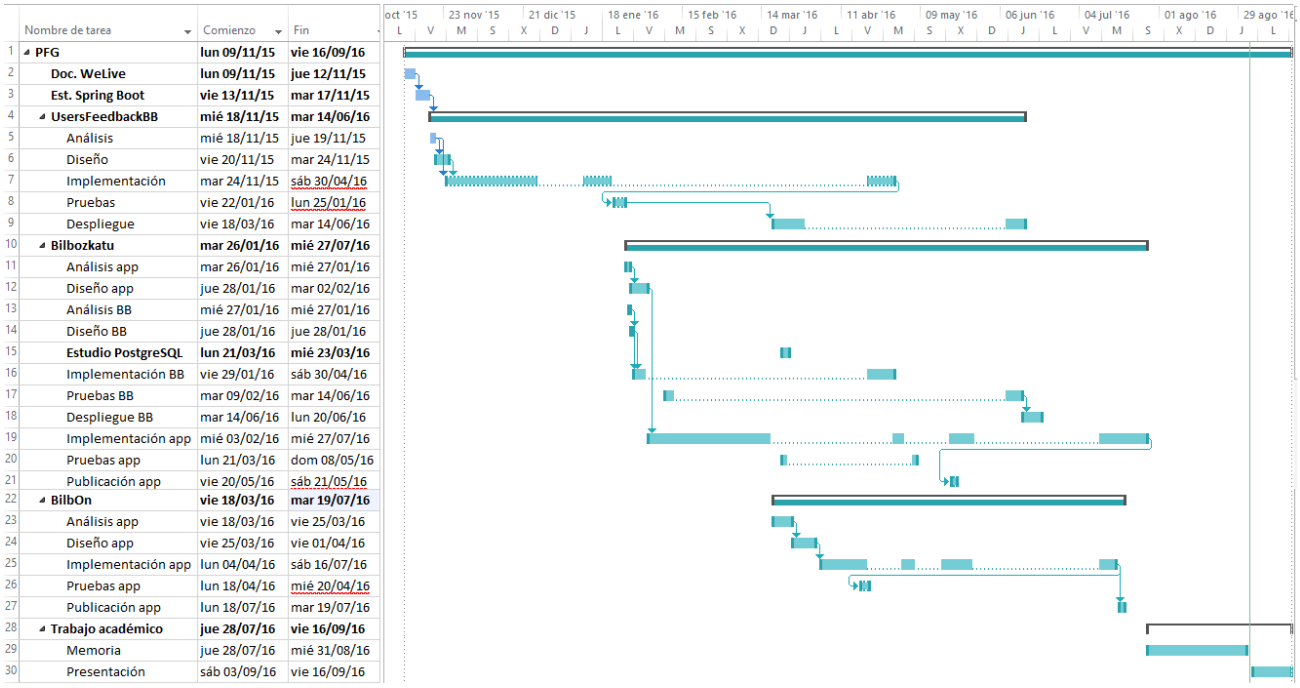


Figura 125: Estructura de Descomposición del Trabajo - EDT (simplificado)

7.5.2. Diagrama de Gantt

En este segundo punto perteneciente a la planificación se muestra el diagrama de Gantt con los plazos que se han ido siguiendo finalmente. En el diagrama se puede apreciar cómo la implementación de las dos apps (tareas 19 y 25 de la Figura 126) no ha sido continua de principio a fin hasta darla por terminada. Al contrario, las aplicaciones no se han podido dar por cerradas por el momento por los siguientes puntos:

- Los cambios en las especificaciones de los componentes de WeLive a integrar requiere de corregir la implementación.
- La necesidad de añadir o mejorar funcionalidades de acuerdo a la opinión de las personas que han realizado un proceso de testing en las apps.
- Las exigencias del cliente, en este caso los responsables del proyecto WeLive y la Comisión Europea, para añadir una serie de funcionalidades en todas las apps de WeLive, como el cuestionario de opinión de los usuarios o las condiciones y términos de uso.



	Nombre de tarea	Comienzo	Fin
1	**PFG**	**lun 09/11/15**	**vie 16/09/16**
2	**Doc. WeLive**	**lun 09/11/15**	**jue 12/11/15**
3	**Est. Spring Boot**	**vie 13/11/15**	**mar 17/11/15**
4	**UsersFeedbackBB**	**mié 18/11/15**	**mar 14/06/16**
5	Análisis	mié 18/11/15	jue 19/11/15
6	Diseño	vie 20/11/15	mar 24/11/15
7	Implementación	mar 24/11/15	sáb 30/04/16
8	Pruebas	vie 22/01/16	lun 25/01/16
9	Despliegue	vie 18/03/16	mar 14/06/16
10	**Bilbozkatu**	**mar 26/01/16**	**mié 27/07/16**
11	Análisis app	mar 26/01/16	mié 27/01/16
12	Diseño app	jue 28/01/16	mar 02/02/16
13	Análisis BB	mié 27/01/16	mié 27/01/16
14	Diseño BB	jue 28/01/16	jue 28/01/16
15	**Estudio PostgreSQL**	**lun 21/03/16**	**mié 23/03/16**
16	Implementación BB	vie 29/01/16	sáb 30/04/16
17	Pruebas BB	mar 09/02/16	mar 14/06/16
18	Despliegue BB	mar 14/06/16	lun 20/06/16
19	Implementación app	mié 03/02/16	mié 27/07/16
20	Pruebas app	lun 21/03/16	dom 08/05/16
21	Publicación app	vie 20/05/16	sáb 21/05/16
22	**BilbOn**	**vie 18/03/16**	**mar 19/07/16**
23	Análisis app	vie 18/03/16	vie 25/03/16
24	Diseño app	vie 25/03/16	vie 01/04/16
25	Implementación app	lun 04/04/16	sáb 16/07/16
26	Pruebas app	lun 18/04/16	mié 20/04/16
27	Publicación app	lun 18/07/16	mar 19/07/16
28	**Trabajo académico**	**jue 28/07/16**	**vie 16/09/16**
29	Memoria	jue 28/07/16	mié 31/08/16
30	Presentación	sáb 03/09/16	vie 16/09/16

Figura 126: Diagrama Gantt del proyecto

En cuanto a las tareas que se muestran en el diagrama se ha hecho una división lógica de acuerdo a los artefactos software que había que desarrollar. No se ha incluido la tarea de formación en Ionic (e implícitamente en AngularJS, etc.) ya que esta se entiende que está incluida en la implementación de Bilbozkatu, a pesar de haberlo indicado por separado en la tabla de costes (Tabla 16). Cabe destacar que el servicio *BilbozkatuBB* y la app Bilbozkatu están dentro de la tarea "Bilbozkatu", ya que se han desarrollado en paralelo.

El proyecto se ha desarrollado principalmente en la empresa Eurohelp, pero en diferentes periodos:

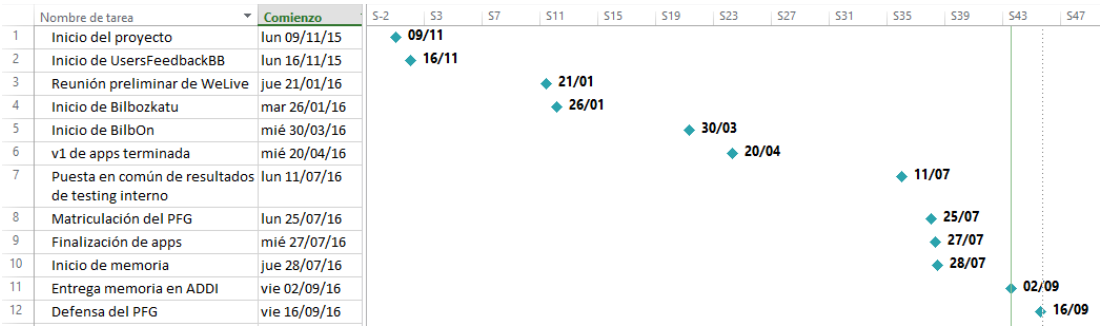
1. Después de unos meses de prácticas en dicha empresa, el primer periodo del proyecto corresponde con la última parte del convenio de prácticas iniciado a mediados de septiembre, con fecha final el 18 de diciembre de 2015. Este periodo de prácticas ha sido compaginado con la realización de las últimas asignaturas de la carrera.
2. El segundo periodo corresponde al convenio de PFG firmado por la empresa con la Universidad del País Vasco, con fecha de inicio el 11 de enero de 2016 y fecha fin el 29 de abril de 2016. No obstante, debido a que correspondían unos días de vacaciones, este periodo terminó el 20 de abril.
3. Por último, a partir del 8 de junio de 2016 hasta la fecha el autor del presente proyecto de fin de grado ya dispone de un contrato de trabajo a tiempo completo. Se ha seguido tanto con el proyecto WeLive como con otros nuevos, lo que ha hecho disminuir el tiempo disponible para la realización de la última parte del proyecto de fin de grado.

Dicho esto, las fechas entre las que se ha desarrollado el proyecto, a falta de la realización de la defensa son:

- Fecha de inicio: 9 de noviembre de 2015
- Fecha fin: 31 de agosto de 2016

7.5.3. Diagrama de hitos

Para terminar se muestra el diagrama de hitos con los acontecimientos más relevantes sucedidos en las 43 semanas del ciclo de vida del presente proyecto de fin de grado.



	Nombre de tarea	Comienzo
1	Inicio del proyecto	lun 09/11/15
2	Inicio de UsersFeedbackBB	lun 16/11/15
3	Reunión preliminar de WeLive	jue 21/01/16
4	Inicio de Bilbozkatu	mar 26/01/16
5	Inicio de BilbOn	mié 30/03/16
6	v1 de apps terminada	mié 20/04/16
7	Puesta en común de resultados de testing interno	lun 11/07/16
8	Matriculación del PFG	lun 25/07/16
9	Finalización de apps	mié 27/07/16
10	Inicio de memoria	jue 28/07/16
11	Entrega memoria en ADDI	vie 02/09/16
12	Defensa del PFG	vie 16/09/16

Figura 127: Diagrama de hitos del proyecto

La reunión preliminar del 21 de enero de 2016 hace referencia a una reunión en empresa con la directora del proyecto para analizar, a grandes rasgos, los artefactos software a desarrollar en el proyecto. Y en cuanto a la reunión con el responsable de WeLive en Eurohelp del 11 de julio, se trataron los resultados o

feedbacks que personal de Eurohelp y Deusto comentaron del proceso de testing de las apps. Es importante que personas ajenas al desarrollo de las apps participen en estas pruebas, y esto hizo que se identificaran una serie de mejoras, algunas más relevantes que otras, que hicieron que hubiera que seguir a partir de la versión 1 de las apps que se había finalizado al término del convenio de PFG el 20 de abril de 2016.

Capítulo 8

8. CONCLUSIONES

En este capítulo 8 se comentan las conclusiones que el autor del presente proyecto de fin de grado extrae en cuanto a objetivos logrados, gestión del proyecto y aspectos personales.

8.1. Conclusiones sobre los objetivos

Desde el inicio del proyecto se definieron una serie de objetivos muy ambiciosos que hicieron que el cumplimiento de los mismos no fuera nada sencillo, incluso se pensó que se habían marcado unas metas un tanto irreales. Hubiera sido algo lógico la necesidad de realizar un proceso de aprendizaje sobre alguna tecnología no vista en la carrera, complementando el uso de otras ya conocidas, para hacer el proyecto de fin de grado.

Sin embargo, la cantidad de tecnologías novedosas, herramientas, y el propio marco que ofrece la realización del proyecto dentro de otro mucho más grande a nivel europeo, en el que colaboran varias empresas de diversos países y financiado por la Comisión Europea, hace que el proyecto desde un inicio haya sido tan complejo como interesante.

Son muchos los meses en los que se ha estado peleando por lograr los objetivos marcados y finalmente se ha conseguido, también en relación a las nuevas exigencias que han ido surgiendo a lo largo del desarrollo del proyecto, como se ha analizado. A partir de ahí, el proyecto continúa y más allá del proyecto de fin de grado, al estar ya trabajando en esa empresa ya se están definiendo nuevas características que se mencionan en el capítulo 9 de propuestas de mejora.

8.2. Conclusiones sobre la gestión

La gestión del proyecto se ha realizado de acuerdo a metodologías ágiles, en este caso SCRUM. Si bien en la carrera se había aplicado esta metodología en alguna que otra asignatura, el haber realizado el proyecto en un entorno laboral ha hecho que la aplicación de SCRUM se haya hecho de manera profesional, aprendiendo mucho de ello y de las relaciones laborales.

De hecho, meses antes de comenzar el proyecto se había estado de prácticas en la empresa, trabajando en otros proyectos. Dado que Eurohelp tiene sede en varias ciudades, en este caso tocó el estar en un proyecto con gente de Bilbao, cuyas reuniones se tenían que hacer por videollamada. El contacto con el mundo laboral implica un aprendizaje muy importante en el uso de buenas prácticas, y en este caso ha resultado de gran utilidad la metodología aplicada, ya que ha facilitado la gestión del proyecto con reuniones de seguimiento.

También ha habido otro factor importante en la gestión del proyecto. Se trata de JIRA, una herramienta para la gestión operativa de proyectos y de incidencias. Gracias a esta herramienta se han ido registrando las trazas de trabajo realizadas para cada tarea especificada, y esto también ha resultado de gran ayuda y aprendizaje en la gestión del proyecto.

8.3. Conclusiones personales

El autor de este proyecto de fin de grado no puede estar más satisfecho del trabajo realizado, aunque haya habido que trabajar mucho. Se ha adquirido una muy importante experiencia de todo lo que el proyecto ha abarcado:

- Desarrollo en un entorno laboral con un equipo de profesionales.
- Aprendizaje y uso de tecnologías punteras dentro de un proyecto de i+D a nivel Europeo, en colaboración con empresas de distintos países.

No sólo el autor del presente documento ha quedado satisfecho, también la empresa. Prueba de ello es la oferta de trabajo recibida poco después de terminar el convenio de PFG, continuando así en el desarrollo de las apps ante la necesidad de integrar nuevas características, tanto comunes a las apps de WeLive como específicas de Bilbozkatu y BilbOn.

El conocimiento adquirido ha sido muy potente. Spring, Spring Boot, Ionic, AngularJS, Apache Cordova, desarrollo de apps híbridas, distintos gestores de paquetes y de versiones, librerías como Google Maps, estudio e integración de componentes de la plataforma WeLive, bases de datos PostgreSQL no analizadas hasta ahora (aunque similares a MySQL), uso de metodologías ágiles en entornos laborales y comunicación con profesionales de otras empresas, todo ello ha supuesto un reto y un aprendizaje que no se esperaba.

La prueba más importante de la confianza que se ha suscitado en la empresa Eurohelp es una que ha sucedido los últimos días a la realización de esta memoria. En la empresa Eurohelp los trabajadores nuevos o que necesita coger experiencia suelen ser ubicados en proyectos de i+D que pueden resultar interesantes, y no dependen tanto de plazos o de clientes específicos. Pues bien, para una oferta pública adjudicada a la empresa que corresponde con un proyecto en el que se debe desarrollar una parte web y una parte móvil, se ha confiado en el autor del presente documento para la realización de dicha app en Ionic, ya que era esta la tecnología ofertada por Eurohelp para el desarrollo de la misma y la que ha supuesto el punto a favor de Eurohelp.

Con los plazos muy ajustados, el cliente está quedando muy satisfecho con la app que se está desarrollando. Todo el conocimiento adquirido se está ya poniendo en práctica y es algo muy importante para el enriquecimiento personal. Por tanto, este proyecto de fin de grado ha supuesto unos beneficios personales muy importantes, y un aprendizaje de cara al futuro.

Capítulo 9

9. PROPUESTA DE MEJORA

Tal y como ya se ha comentado, el proyecto WeLive está en constante evolución. Procesos de testing internos, nuevas necesidades y exigencias por parte de la Comisión Europea mantienen vivas las aplicaciones, y ya hay una serie de mejoras definidas que se van a realizar. Estas propuestas de mejora se analizarán a continuación.

Las posibles mejoras a realizar son:

- Implementar, en Bilbozkatu, un sistema de moderación de comentarios y propuestas, a fin de que un usuario no pueda introducir cualquier tipo de propuesta o comentario y automáticamente ser visto por el resto de usuarios.

  Para ello, se debe hacer uso del concepto de roles para los usuarios, de manera que un usuario con los privilegios suficientes pueda visualizar las propuestas y comentarios que están pendiente de aprobación.

- Añadir la opción de subir una imagen adjunta al crear una propuesta en la app de Bilbozkatu.
- Almacenar el idioma seleccionado como preferencia del usuario en la app. De este modo el idioma cargado al iniciar la app, tanto Bilbozkatu como BilbOn, será el último utilizado en la misma.
- En BilbOn, implementar un servicio de valoración de POIs, además de permitir enviar comentarios.
- Otros detalles:
  - Los logos que se muestran en el *splash screen* sólo pueden ser visualizados en versiones de Android mayores a la 4.1. Esto es debido a que el diseñador gráfico que los ha hecho los ha generado como *svg* y no como imágenes simples. El soporte de *svg* en versiones inferiores a la 4.1 de Android es limitado.
  - Estructurar mejor los ficheros en donde se guardan las traducciones al euskara, castellano e inglés. Utilizar el formato JSON para una mayor legibilidad de dichos recursos.

Las aplicaciones de WeLive requieren de la aprobación de la Comisión Europea y ésta realiza periódicamente análisis y verificaciones de las mismas. De este modo se implementa un modelo de desarrollo que hace que la aplicación esté viva y en constante evolución.

Capítulo 10

10. BIBLIOGRAFÍA

En este apartado se muestran las referencias bibliográficas más relevantes en relación al proyecto de fin de grado realizado. Evidentemente no se muestran algunas secundarias consultadas puntualmente para la formación y uso de las tecnologías en el desarrollo de los distintos artefactos software.

[1] Página oficial del proyecto WeLive
© 2016 WeLive. http://www.welive.eu/

[2] Documentación del Framework de Ionic y todos sus componentes
Code licensed under MIT. © 2013-2016. http://ionicframework.com/

[3] Documentación para desarrolladores de AngularJS (CC BY 3.0).
Code licensed under The MIT License. https://docs.angularjs.org/guide

[4] Documentación de Apache Cordova y plugins
Copyright © 2012, 2013, 2015 The Apache Foundation, licenciado bajo Apache License Version 2.0. https://cordova.apache.org/docs/en/latest/

[5] Librería de plugins de Apache Cordova para AngularJS
http://ngcordova.com/docs/plugins/

[6] Spring Framework
© 2016 Pivotal Software, Inc- All Rights Reserved. https://spring.io/

[7] Guía de referencia de Spring Boot
Copyright © 2013 - 2016 http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

[8] Documentación oficial de *Spring Web MVC Framework*
http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

[9] *Raman Kazhamiakin, Gabriele Zacco, Marco Pistore: WeLive's AAC User Guide, 2015*

[10] *Raman Kazhamiakin, Gabriele Zacco, Marco Pistore: WeLive's Logging Building Block User Guide, 2016*

[11] *Dr. M. Elkstein*: Tutorial de servicios web de estilo REST
http://rest.elkstein.org/

[12] Documentación de Cloud Foundry
http://docs.cloudfoundry.org/

[13] Documentación de la API de Google Maps para JavaScript
https://developers.google.com/maps/documentation/javascript/

[14] Documentación del sistema gestor de versiones Git
https://git-scm.com/doc

[15] Gestor de paquetes JavaScript Bower
https://bower.io/

[16] Foro de desarrolladores Stack Overflow
http://stackoverflow.com/

[17] *Marc Hadley, Sun Microsystems Inc: Web Application Description Language*
https://www.w3.org/Submission/wadl/

[18] Tutorial de PostgreSQL
Copyright © 2016 http://www.postgresqltutorial.com/

[19] Documentación official de PostgreSQL 9.0.23
©1996-2015 https://www.postgresql.org/docs/9.0/static/index.html

Capítulo 11

11. GLOSARIO Y ACRÓNIMOS

En este último apartado se muestran las definiciones de algunos de los términos o acrónimos presentes en el documento, a fin de facilitar al lector la total comprensión del mismo.

AAC BB (*Authentication and Authorization Control System Building Block*)**:** Es el componente de la plataforma WeLive que se encarga de la gestión de los usuarios de WeLive. Ofrece una API mediante la cual las aplicaciones pueden, por ejemplo, obtener los *tokens* de usuarios, a fin de acceder a recursos protegidos del entorno WeLive.

API (*Application Programing Interface*): Es una interfaz de programación de aplicaciones a través de la cual un software puede utilizar las funciones, procedimientos y subrutinas de una cierta biblioteca.

Android: Android es un sistema operativo para móviles desarrollado por Google y diseñado principalmente para dispositivos móviles táctiles como *smartphones* y *tablets*. Es uno de los sistemas operativos más utilizados del mercado.

Android SDK (*System Development Kit*): El kit de desarrollo del sistema de Android es un conjunto de herramientas de desarrollo usadas para desarrollar aplicaciones para la plataforma Android. En este contexto, este kit es utilizado a para compilar el código de una app desarrollada con tecnologías web como una aplicación nativa de Android.

App móvil híbrida: Es una aplicación para dispositivos móviles desarrollada con tecnologías web como HTML, CSS y JavaScript, que permite la compilación de la misma para diferentes plataformas móviles como Android o iOS. Para ello, estas apps están almacenadas en una aplicación nativa que utiliza el *Web View* de las plataformas de móviles para su visualización.

App móvil nativa: Es una aplicación para dispositivos móviles desarrollada con lenguajes de programación específicos de cada plataforma (como *Objective C* para iOS o Java para Android). Ofrecen un gran rendimiento, pero el coste de desarrollo de una app para diferentes plataformas es mucho mayor que con las aplicaciones híbridas.

Building Block (BB): En el contexto del proyecto WeLive, un *building block* es un componente que accede a un grupo de *datasets* o conjuntos de datos. De esta manera, aísla a los desarrolladores de aplicaciones de la idiosincrasia de gestionar el acceso a *datasets* concretos que pueden estar en cualquier tipo de formato o ubicación (RDF, datos de redes sociales, datos del sector privado…).

CLI *(Command Line Interface):* Una interfaz de línea de commandos es una herramienta que permite a un usuario interactuar con un programa informático o aplicación, escribiendo para ello instrucciones a través de una línea de texto simple.

Cloud Foundry: Es un "servicio como plataforma" o *Platform as a Service* (PaaS) de código libre en la nube, que provee todo lo necesario para el despliegue de aplicaciones en la nube. Abstrae al usuario de toda la infraestructura necesaria así como de la portabilidad entre *Clouds,* entre otros.

Dataset: En el contexto del proyecto WeLive, un *dataset* de servicio público es un conjunto de datos que puede ser de interés para la creación de nuevas apps públicas. Pueden pertenecer tanto a la Administración Pública como a servicios externos como *OpenStreetMap* o sectores privados de una compañía en concreto.

IDE (*Integrated Development Environment*): Es una aplicación software que provee de un entorno de desarrollo en el que se integran facilidades para el desarrollo de software por parte de los programadores informáticos. Generalmente consiste en un editor de código, herramientas para la compilación y un *debugger*, un programa informático usado para testear y probar los programas desarrollados.

Git: Es un software de control de versiones pensado en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

Google Play: Anteriormente conocido como *Android* Market, es una plataforma para Android creada por Google para centralizar aplicaciones o contenidos multimedia. Principalmente se utiliza como una tienda de aplicaciones.

HTTP (*Hypertext Transfer Protocol*): Es un protocolo de transferencia de hipertexto con el que se permiten las transferencias de información en la *World Wide Web*. Se emplea en la capa de transporte de la arquitectura TCP.

JSON (*JavaScript Object Notation*): Es un formato ligero de intercambio de datos, organizado por pares clave-valor. Es entendible por humanos e interpretable por máquinas.

JUnit: Es un *framework* con el que poder realizar pruebas unitarias sobre un programa informático a través de clases Java. De esta manera se puede verificar el comportamiento de componentes software.

Java Config: En aplicaciones Java, como las desarrolladas con Spring, Java Config es un modo fácil y seguro de acceder a las propiedades de configuración normalmente definidas en ficheros *.properties*.

Linked Data: *Linked Data* o datos enlazados es la forma que tiene la web semántica de vincular los distintos datos que están distribuidos en la red, de forma que se referencia de la misma forma que lo hacen los enlaces de las páginas web. Gracias a esto, partiendo de un origen, las personas y máquinas pueden descubrir nueva información, gracias a que los datos están interconectados y distribuidos en la web como una gran base de datos.

ODS (*Open Data Stack*): En el contexto del proyecto WeLive, es un componente que trata con los desafíos asociados a la gestión del conocimiento de una ciudad en la forma de múltiples orígenes de datos con distintos formatos. Provee una API mediante la cual acceder fácilmente a dichos *datasets*.

Open Government: Es una doctrina política catacterizada por la adopción de la filosofía del movimiento del software libre a los principios de la democracia. Tiene como objetivo que los ciudadanos colaboren en la creación y mejora de servicios públicos.

Plataforma WeLive: Es el entorno en el que se enmarcan los múltiples componentes de WeLive (los *Building Blocks,* componentes, aplicaciones, etc.).

Plugin: En informática, un plugin es un programa informático o aplicación que se relaciona o inyecta con otra para dotarla de nueva funcionalidad. En el contexto de Cordova, permite gestionar características nativas de los dispositivos pertenecientes a diferentes plataformas como Android e iOS mediante un desarrollo híbrido (genérico para cualquier plataforma).

Página web responsiva: Es una web diseñada con *Responsive Web Design* o diseño responsivo, es decir, una web cuyo contenido se adapta a los tamaños de pantallas en las que se abre de forma dinámica, ofreciendo una experiencia de visita óptima.

RDF (*Resource Description Framework*): Es una familia de especificaciones del consorcio de la *World Wide Web***,** con la que se ofrece un método para expresar el conocimiento en un mundo descentralizado. Es el fundamento de la web semántica, en la que las aplicaciones informáticas utilizan información estructurada distribuida por toda la web.

REST (*Representational State Transfer*): Es un estilo de arquitectura para servicios web sin estado, es decir, en la que cada llamada es independiente (por lo que es el usuario el que debe indicar siempre quién es en caso de necesitar autenticación, por medio de parámetros o algún *token*).

SPARQL (*SPARQL Protocol RDF and Query Language*): es el lenguaje estandarizado utilizado para la consulta de grafos RDF, enmarcado dentro de la web semántica.

Servlet: En Java un *servlet* es una clase utilizada para ampliar las capacidades de un servidor. Generalmente se utilizan para extender las aplicaciones alojadas por servidores web, de tal manera que pueden ser vistos como *applets* de Java que se ejecutan en servidores en vez de navegadores web. Este tipo de servlets son la contraparte Java de otras tecnologías de contenido dinámico Web, como PHP y ASP.NET.

Smart City: Las *Smart Cities* o ciudades inteligentes se refieren a un tipo de desarrollo urbano basado en la sostenibilidad, que es capaz de responder adecuadamente a las necesidades básicas de instituciones, empresas y de los propios habitantes.

Spring Annotation: En el marco de trabajo de Spring, un *annotation* o anotación es una forma de configurar muchos de los aspectos o componentes que integra, en lugar de utilizar configuraciones basadas en XML.

Sprint: En la metodología de trabajo SCRUM un *sprint es* un periodo de duración constante (definido al inicio del proyecto) que establece el tiempo disponible para generar un entregable. Así, el desarrollo de un proyecto está dividido en *sprints*.

Subversion: Subversion o *SVN e*s una herramienta de control de versions *open source* de Apache, basada en repositorio cuyo funcionamiento se asemeja mucho al de un sistema de ficheros.

Swagger Interface: *Swagger* es un framework para APIs RESTful que ayuda en la generación de documentación y la relaciona automáticamente con la implementación, proporcionando también una representación interactiva de la API, llamada *Swagger Interface,* que permite un fácil testeo de las llamadas sin tener que hacerlo desde código en una app.

TIC: Hace referencia a las Tecnologías de la Información y Comunicación, caracterizadas por la digitalización de las tecnologías de registros de contenidos como la informática, comunicaciones y telemática.

Token: Un *token* conocido como *token* de seguridad, de autenticación o criptográfico, es un dispositivo electrónico que se le ofrece a un usuario autorizado de un servicio informático para facilitar el proceso de autenticación.

Web View: Es un navegador simple perteneciente a un sistema operativo móvil como Android que muestra contenido web directamente dentro de una app.

ANEXO A: BB’s – Diagrama de clases extendido

En este Anexo A se muestra el diagrama de clases extendido perteneciente al diseño de los *Building Blocks* desarrollados (en los apartados 6.1.2.3 y 6.2.2.3 se realizan las explicaciones pertinentes del diagrama de clases).

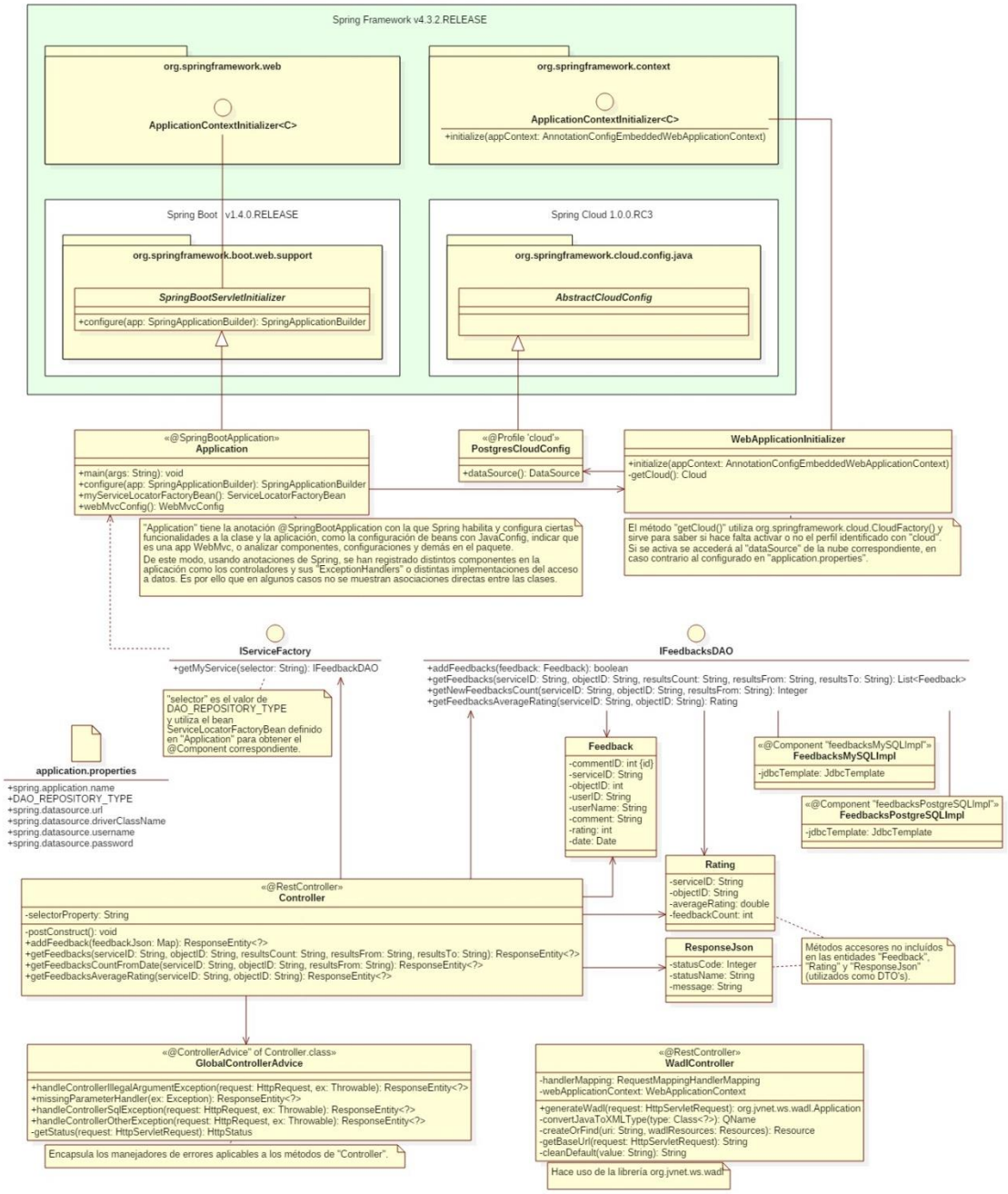


Figura 128: Diagrama de Clases extendido de UsersFeedackBB

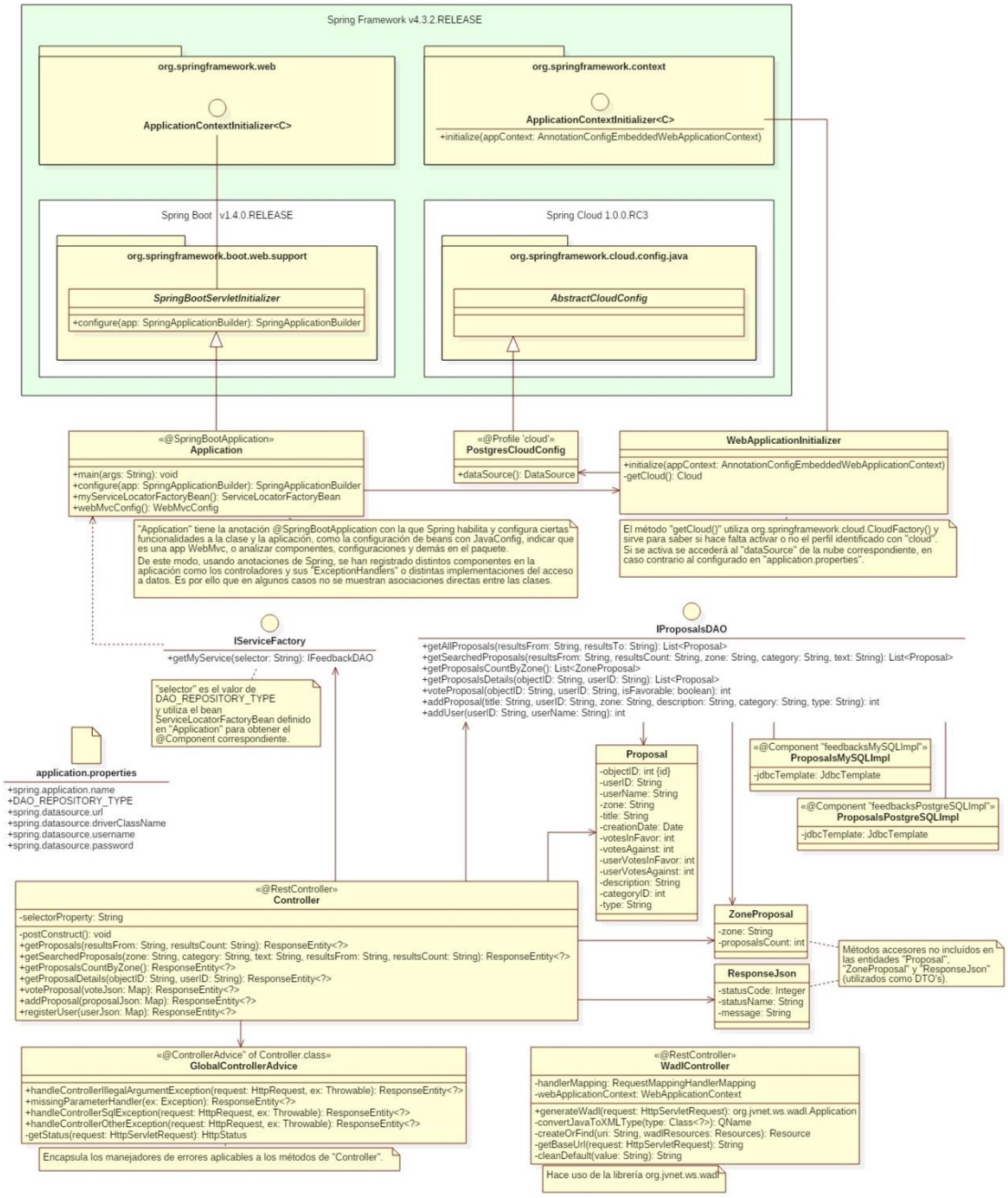


Figura 129: Diagrama de Clases extendido de BilbozkatuBB

Además de las clases mostradas en los diagramas del capítulo 6, en las Figuras 128 y 129 se puede apreciar el uso de algunas clases del *framework* Spring, y del componente Spring Boot dentro de él. Estas clases se han utilizado para el arranque e inicialización de la aplicación, y en caso de ejecutarse en la nube (en la plataforma Cloud Foundry) se hace uso de la clase de configuración *PostgreCloudConfig* que extiende la clase *AbstractCloudConfig* del componente Spring Cloud necesario. Con ello se define el *DataSource* configurado en dicho *PostgreCloudConfig*, que es el asociado a la aplicación en Cloud Foundry.

ANEXO B: BB’s - Scripts SQL del Modelo de datos

En este Anexo B se muestran los *scripts* utilizados para la creación del modelo de datos de *UsersFeedbackBB* y *BilbozkatuBB*, tanto para el soporte MySQL como para el de PostgreSQL que se incluyó posteriormente. En este segundo caso se ha tratado de generar un modelo equivalente teniendo en cuenta las peculiaridades y diferencias en los tipos de datos entre ambas bases de datos.

UsersFeedbackBB

La creación de la base de datos para *UsersFeedbackBB* y de la tabla ‘comment’ mostrada en el punto 6.1.2.5 se realiza de la siguiente manera.

```
CREATE DATABASE  IF NOT EXISTS `users_feedback`;
USE `users_feedback`;

CREATE TABLE IF NOT EXISTS `comment` (
  `commentID` int(11) unsigned NOT NULL AUTO_INCREMENT COMMENT 'INT(11): display integer values
                                       having a width less than the width specified (11) for
                                       the column by left-padding them with spaces. ',
  `serviceID` varchar(45) NOT NULL,
  `objectID` int(11) NOT NULL,
  `userID` varchar(45) NOT NULL,
  `userName` varchar(45) NOT NULL,
  `comment` longtext NOT NULL,
  `rating` tinyint(2) unsigned NOT NULL,
  `date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`commentID`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8
  COMMENT='Store the citizen''s feedbacks about some services.';

SELECT * FROM users_feedback.comment;
```

Figura 130: Script MySQL de creación del modelo de datos para UsersFeedbackBB

Sin embargo, en el caso de una base de datos PostgreSQL esto se ha tenido que hacer de otro modo. Dentro de una misma base de datos se han creado diferentes esquemas o *schemas*, cada uno de los cuales corresponde a un *Building Block*. La creación de la base de datos en PostgreSQL se hace de este modo (ver Figura 131).

```
-- Database: welive
-- DROP DATABASE welive;
CREATE DATABASE welive
  WITH OWNER = welive
       ENCODING = 'UTF8'
       TABLESPACE = pg_default
       LC_COLLATE = 'Spanish_Spain.1252'
       LC_CTYPE = 'Spanish_Spain.1252'
       CONNECTION LIMIT = -1;

COMMENT ON DATABASE welive
  IS 'connection database for WeLive';
```

Figura 131: Script de creación de BD para PostgreSQL

En la anterior Figura 131 se muestra que el propietario de la base de datos de nombre 'welive' es el usuario llamado 'welive' también. Este usuario se mostrará en los siguientes *scripts* en la creación de los distintos *schemas* y tablas en los *schemas*. Dicho esto, la creación del modelo de datos para *UsersFeedbackBB* en PostgreSQL se ha hecho tal y como se muestra en la siguiente Figura 132.

```
-- Schema: usersfeedbackbb
-- DROP SCHEMA usersfeedbackbb;
CREATE SCHEMA usersfeedbackbb
  AUTHORIZATION welive;

GRANT ALL ON SCHEMA usersfeedbackbb TO welive;
COMMENT ON SCHEMA usersfeedbackbb
  IS 'schema for usersFeedbackBB Building Block';




-- Table: usersfeedbackbb.comment
-- DROP TABLE usersfeedbackbb.comment;
CREATE TABLE usersfeedbackbb.comment
(
  commentid serial NOT NULL, -- unique identifier column (similar to the AUTO_INCREMENT
                             -- property supported by some other databases)
  serviceid character varying(45) NOT NULL,
  objectid integer NOT NULL,
  userid character varying(45) NOT NULL,
  username character varying(45) NOT NULL,
  comment text NOT NULL,
  rating smallint NOT NULL,
  date timestamp with time zone NOT NULL DEFAULT now(),
  CONSTRAINT "commentID_PK" PRIMARY KEY (commentid)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE usersfeedbackbb.comment
  OWNER TO welive;
COMMENT ON TABLE usersfeedbackbb.comment
  IS 'This table stores the citizen''s feedbacks about something';
COMMENT ON COLUMN usersfeedbackbb.comment.commentid
  IS 'unique identifier column (similar to the AUTO_INCREMENT
      property supported by some other databases)';
```

Figura 132: Script PostgreSQL de creación del modelo de datos para UsersFeedbackBB

Una vez creada la base de datos, se crea el *schema* y en él la tabla 'comment', como se ha hecho para MySQL. Los tipos de datos utilizados son equivalentes a los empleados en la base de datos MySQL, a fin de implementar lo más parecido posible los distintos soportes para las bases de datos (es decir, los DAOs) de los *Building Blocks*.

A continuación se muestran los *scripts* utilizados para crear el modelo de *BilbozkatuBB* para MySQL y PostgreSQL. De nuevo, en el primer caso esto se realiza directamente, creando las tablas necesarias en la nueva base de datos, y en el segundo se crea un nuevo esquema en la base de datos antes mencionada de la Figura 131.

BilbozkatuBB

```
CREATE DATABASE IF NOT EXISTS `welive`;
USE `welive`;

CREATE TABLE IF NOT EXISTS `user` (
  `userID` varchar(45) NOT NULL,
  `userName` varchar(45) NOT NULL,
  PRIMARY KEY (`userID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

SELECT * FROM welive.user;


CREATE TABLE IF NOT EXISTS `proposal` (
  `objectID` int(11) unsigned NOT NULL AUTO_INCREMENT COMMENT 'INT(11): display
                                    integer values having a width less than
                                    the width specified (11) for the column by
                                    left-padding them with spaces. ',
  `userID` varchar(45) NOT NULL,
  `title` varchar(200) NOT NULL,
  `zone` varchar(60) NOT NULL,
  `description` longtext NOT NULL,
  `category` int(11) unsigned NOT NULL,
  `type` enum('ciudadano') NOT NULL,
  `date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`objectID`),
  KEY `FK_UserID_idx` (`UserID`),
  CONSTRAINT `FK_Proposal_UserID`
      FOREIGN KEY (`userID`) REFERENCES `user` (`userID`)
      ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

SELECT * FROM welive.proposal;


CREATE TABLE IF NOT EXISTS `vote` (
  `objectID` int(11) unsigned NOT NULL,
  `userID` varchar(45) NOT NULL,
  `isFavorable` tinyint(1) NOT NULL COMMENT 'A value of zero is considered false.
                                    Non-zero values are considered true.',
  `date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`objectID`,`userID`),
  KEY `FK_UserID_idx` (`userID`),
  CONSTRAINT `FK_ObjectID`
      FOREIGN KEY (`objectID`) REFERENCES `proposal` (`objectID`)
      ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `FK_UserID`
      FOREIGN KEY (`userID`) REFERENCES `user` (`userID`)
      ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

SELECT * FROM welive.vote;
```

Figura 133: Script MySQL de creación del modelo de datos para BilbozkatuBB

El *script* de la Figura 133 se corresponde con el modelo analizado en el apartado 6.2.2.5, concretamente en la Figura 48. No tiene nada complejo que explicar, ya que tan sólo se definen las tablas con los correspondientes campos y restricciones de clave primaria y de clave extranjera allá donde son necesarias. En este caso se han creado las tablas 'proposal', 'user' y 'vote', de acuerdo al diseño realizado.

A continuación se muestra el esquema creado para albergar el modelo de *BilbozkatuBB* en la base de datos PostgreSQL antes creada, y cada una de las tres tablas de dicho modelo.

```
-- Schema: bilbozkatubb
-- DROP SCHEMA bilbozkatubb;
CREATE SCHEMA bilbozkatubb
  AUTHORIZATION welive;

GRANT ALL ON SCHEMA bilbozkatubb TO welive;
COMMENT ON SCHEMA bilbozkatubb
  IS 'schema for bilbozkatuBB Building Block';




-- Table: bilbozkatubb."user"
-- DROP TABLE bilbozkatubb."user";
CREATE TABLE bilbozkatubb."user"
(
  userid character varying(45) NOT NULL,
  username character varying(45) NOT NULL,
  CONSTRAINT "userID_PK" PRIMARY KEY (userid)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE bilbozkatubb."user"
  OWNER TO welive;
COMMENT ON TABLE bilbozkatubb."user"
  IS 'This table stores the users';
```

Figura 134: Script PostgreSQL para el esquema *y tabla 'user' de BilbozkatuBB*

```
-- Table: bilbozkatubb.vote
-- DROP TABLE bilbozkatubb.vote;
CREATE TABLE bilbozkatubb.vote
(
  objectid integer NOT NULL,
  userid character varying(45) NOT NULL,
  isfavorable boolean NOT NULL, -- Valid literal values for the "true" state are:
                                --     true, y, yes, on and 1 as strings, or TRUE.
                                -- For the "false" state, the following values can be used:
                                --     false, f, no, off and 0 as strings, or FALSE
  date timestamp with time zone NOT NULL DEFAULT now(),
  CONSTRAINT "objectID_userID_PK" PRIMARY KEY (objectid, userid),
  CONSTRAINT "objectID_FK" FOREIGN KEY (objectid)
      REFERENCES bilbozkatubb.proposal (objectid) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT "userID_FK" FOREIGN KEY (userid)
      REFERENCES bilbozkatubb."user" (userid) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT vote_objectid_check CHECK (objectid >= 0)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE bilbozkatubb.vote
  OWNER TO welive;
COMMENT ON TABLE bilbozkatubb.vote
  IS 'This table stores the users'' votes of each proposal';
COMMENT ON COLUMN bilbozkatubb.vote.isfavorable
IS 'Valid literal values for the "true" state are:
    true, y, yes, on and 1 as strings, or TRUE.
    For the "false" state, the following values can be used:
    false, f, no, off and 0 as strings, or FALSE';
```

Figura 135: Script PostgreSQL para la tabla 'vote' de BilbozkatuBB

```
-- Table: bilbozkatubb.proposal
-- DROP TABLE bilbozkatubb.proposal;
CREATE TABLE bilbozkatubb.proposal
(
  objectid serial NOT NULL, -- unique identifier column (similar to the AUTO_INCREMENT
                            -- property supported by some other databases)
  userid character varying(45) NOT NULL,
  title character varying(200) NOT NULL,
  zone character varying(60) NOT NULL,
  description text NOT NULL,
  category integer NOT NULL,
  type text NOT NULL,
  date timestamp with time zone NOT NULL DEFAULT now(),
  CONSTRAINT "objectID_PK" PRIMARY KEY (objectid),
  CONSTRAINT "userID_FK" FOREIGN KEY (userid)
      REFERENCES bilbozkatubb."user" (userid) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT proposal_category_check CHECK (category >= 0),
  CONSTRAINT proposal_type_check CHECK (lower(type) = 'ciudadano'::text)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE bilbozkatubb.proposal
  OWNER TO welive;
COMMENT ON TABLE bilbozkatubb.proposal
  IS 'This table stores the users'' proposals';
COMMENT ON COLUMN bilbozkatubb.proposal.objectid
IS 'unique identifier column (similar to the AUTO_INCREMENT
    property supported by some other databases)';

-- Index: bilbozkatubb.proposal_lower_title_idx
-- DROP INDEX bilbozkatubb.proposal_lower_title_idx;
CREATE INDEX proposal_lower_title_idx
  ON bilbozkatubb.proposal
  USING btree
  (lower(title::text) COLLATE pg_catalog."default" varchar_pattern_ops);

-- Index: bilbozkatubb.proposal_lower_description_idx
-- DROP INDEX bilbozkatubb.proposal_lower_description_idx;
CREATE INDEX proposal_lower_description_idx
  ON bilbozkatubb.proposal
  USING btree
  (lower(description) COLLATE pg_catalog."default" varchar_pattern_ops);
```

Figura 136: Script PostgreSQL para la tabla 'proposal' de BilbozkatuBB

En esta última tabla, 'proposal', se muestran los índices que se han creado. Esto se ha realizado para mejorar el rendimiento al realizar la búsqueda de propuestas filtrando por título o descripción. Y es que al implementar esa búsqueda se ha utilizado la función *lower()* para que sea insensible a mayúsculas, y esto podría haber tenido cierto coste en cuanto a rendimiento. Gracias a los índices creados esto ya no supone ningún problema.

ANEXO C: Gestión de dependencias

En el desarrollo de software la utilización de librerías externas está a la orden del día para facilitar en la medida de lo posible el trabajo a realizar. En ocasiones, no obstante, la integración de dichas librerías no es algo tan trivial, habiendo dependencias que gestionar. Las bibliotecas utilizadas pueden depender a su vez de otras, y así sucesivamente. Para no tener que preocuparse de esto existen herramientas muy útiles que realizan el trabajo de manera automática, obteniendo todo lo que una librería necesita.

Apache Maven

En el punto 5.3 se ha mencionado la herramienta Apache Maven como la utilizada para gestionar las dependencias de los servicios web implementados. Además de gestionar las dependencias de las librerías que se quieran inyectar o utilizar, sirve también para gestionar proyectos Java en general. En concreto, dispone de un fichero llamado *pom.xml* (del inglés *Project Object Model*, Modelo de Objeto de Proyecto) en el que se pueden configurar numerosos parámetros del proyecto amén de las dependencias.

Tal sólo hay que insertar un pequeño bloque XML en la sección <*dependencies*/> del mismo para añadir una librería al proyecto. Automáticamente, al actualizar el proyecto con Maven, se descargan todas las dependencias necesarias y no hay que preocuparse de nada más. La configuración en el *pom.xml* puede ser más compleja pero no se entra en detalles. Si se utiliza Maven en un entorno de desarrollo como Eclipse también es posible realizar la configuración de manera visual sin tener que escribir nada en formato XML. Las dependencias que se pueden utilizar se pueden encontrar, por ejemplo, en el repositorio central de Maven: http://search.maven.org/.

En el proyecto de fin de grado, pues, se ha hecho uso de esta herramienta para los servicios web implementados (Building Blocks), obteniendo así de manera sencilla todas las dependencias necesarias para, por ejemplo, JUnit, conectores de bases de datos, Spring Framework y Spring Boot, Spring Cloud y demás. De lo contrario se tendrían que haber gestionado los .jar a incluir, tal y como se ha realizado a lo largo de la carrera en algunos proyectos.

Bower.io

Por último, si bien no proporciona tantas características como Apache Maven, se ha hecho uso del gestor de paquetes Bower para gestionar las librerías JavaScript utilizadas (*plugins* de Ionic, de AngularJS o de tratamientos de fechas, entre otros) en las apps. De esta manera esta herramienta se encarga de verificar que las versiones de las librerías utilizadas son compatibles entre sí.