Enfoque reutilizable para el desarrollo de aplicaciones de comercio electrónico: Un caso de estudio basado en el uso de Beacons.

Reusable approach to the development of e-commerce applications: A case study based on the use of Beacons.

Realizado por
Ezequiel Rodríguez Márquez

Tutorizado por
Mónica Pinto Alarcón

Departamento
Lenguajes y Ciencias de la Comunicación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DEL 2020

Fecha defensa: julio de 2020

Resumen

Este proyecto surge debido al auge que hoy en día tienen los dispositivos del Internet de las Cosas (IoT, de *Internet Of Things)*, que nos facilitan distintos aspectos de nuestra vida cotidiana. Un tipo de aplicaciones concreto que puede beneficiarse del uso de estos dispositivos son las aplicaciones del comercio electrónico. Hoy en día, la mayoría de comercios tienen programas de fidelización con sus clientes, donde acceden a descuentos y ofertas a través de su correo electrónico y en muchos casos el cliente no termina beneficiándose de ellas por simple olvido. Mediante el uso de dispositivos de la IoT es posible mejorar estos programas de fidelización avisando al cliente sobre los descuentos que tiene disponible justo en el momento en el que dicho cliente se encuentra visitando el comercio. Por otro lado, puede aumentar las ventas del negocio. Un ejemplo puede ser avisando al cliente mientras se encuentra en la tienda de productos que compra habitualmente o enviándole una oferta personalizada para él con una duración determinada (ej. próximas dos horas, día actual, etc.) para incentivarlo a que compre dichos productos o dándole información del tiempo de retraso estimado sobre una cita, etc. Un dispositivo concreto que está siendo utilizado ampliamente en el comercio son los Beacons. Se trata de un dispositivo IoT basado en la tecnología Bluetooth Low Energy (BLE) que es capaz de enviar señales a dispositivos móviles cercanos a él, previa instalación de la aplicación móvil del comercio y autorización del propietario del dispositivo.

Estas aplicaciones de comercio electrónico que hacen uso de los dispositivos Beacons comparten una serie de funcionalidades comunes que pueden ser reutilizadas entre sí y, por tanto, no tiene sentido desarrollar cada una de estas aplicaciones desde cero. El objetivo principal de este proyecto es el desarrollo de una estructura común reutilizable para el desarrollo de aplicaciones de comercio electrónico basadas en el uso de dispositivos Beacons.

Palabras clave: **Beacon, IoT, Bluetooth, E-commerce.**

Abstract

The following project appears as a result of IoT devices that facilitate different aspects of our daily lives. A specific type of applications that can benefit from the use of these devices are e-commerce applications. Nowadays, most businesses have loyalty programs with their customers, where they access discounts and offers through their email and in many cases the customer does not end up benefiting from them simply by forgetting them. Through the use of IoT devices, it is possible to improve these loyalty programs by notifying the customer about the discounts available to them right at the time when the customer is visiting the store. On the other hand, it can increase the sales of the business, an example can be, advising the customer while he is in the product store where he usually buys or sending him a personalized offer for him with a certain duration (e.g. next two hours, current day , etc.) to encourage you to buy these products or giving you information about the estimated delay time about an appointment, etc.

One specific device that is being widely used in commerce is the Beacons. It is an IoT device based on Bluetooth Low Energy (BLE) technology that is capable of sending signals to mobile devices close to it, after installing a mobile application from the merchant and authorization from the device owner.

The main objective of this project is the development of a common reusable infrastructure for the development of electronic commerce applications based on the use of Beacon devices.

Keywords: **Beacon, IoT, Bluetooth, e-commerce.**

Índice

1

Introducción

En el presente capítulo se van a explicar antecedentes, motivación y objetivos principales del proyecto desarrollado, además de la explicación de la estructura de la memoria.

1.1 Antecedentes

El desarrollo de este proyecto viene dado por la expansión del comercio electrónico. Cada vez son más las compañías que deciden incorporar tecnología para sus comercios, ya sea pedido online y recogida en tienda, promociones online y descuentos entre otros.

Uno de los sectores más beneficiado por el uso de las tecnologías es el comercio electrónico (o *e-commerce* en inglés), que hace hoy en día un uso muy amplio de aplicaciones móviles (*apps*) para fidelizar a los usuarios, por lo que existe una gran relación con los clientes que cada vez se encuentran más satisfechos con sus compras. Esto deja claro que la forma de consumir ha cambiado y las empresas necesitan adaptarse a estos cambios.

El uso de los *beacons* hace que combinado con el comercio electrónico se puedan realizar infinidad de aplicaciones con distintas posibilidades para mejorar la calidad del consumo de los clientes.

1.2 Motivación

Existen varios motivos los cuales llevaron a la realización del presente proyecto. Uno de ellos es el atractivo de usar un dispositivo IoT como Beacon, el cual tiene infinidad de usos y nos facilita muchos aspectos de la vida cotidiana.

Otro de los motivos es la realización de la estructura que no solo sea para mí, sino que sea reutilizable por cualquier persona, aportando mi pequeño grano de arena a la comunidad.

El tercer motivo es Android Studio, ya que no lo había usado nunca y tiene un gran potencial, donde el 87% de dispositivos del mercado español son Android. Por fin lo he podido explorar y aprender lo necesario para la realización de este proyecto.

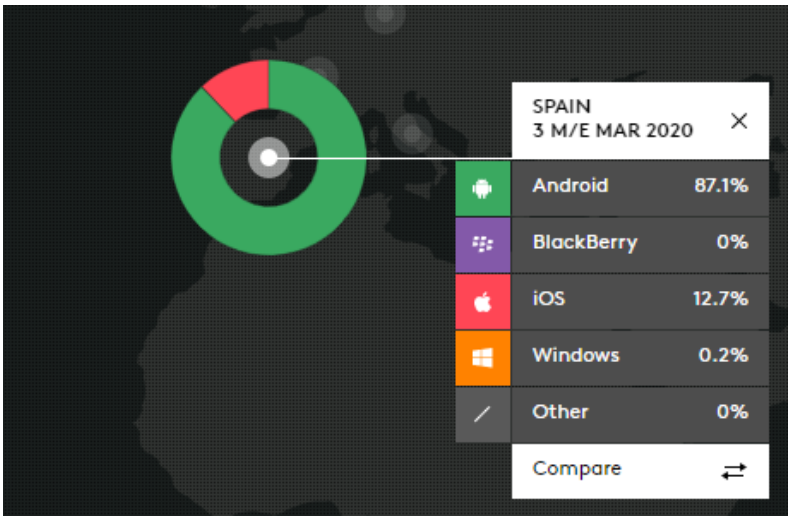

Figura 1. Cuota representativa de sistemas operativos en España. (Imagen tomada de la web Kantar Worldpanel ComTech. - https://www.kantarworldpanel.com/global/smartphone-os-market-share/)

1.2 Objetivos

El gran uso de las tecnologías IoT en el presente hace que el desarrollo de un proyecto de fin de grado que haga uso de estas tecnológicas sea muy interesante para ayudar a terminar la formación recibida por el alumno durante la titulación. Para profundizar en el estudio y uso de estas tecnologías, el objetivo principal de este proyecto es el:

Desarrollo de una estructura común reutilizable para el desarrollo de las aplicaciones de comercio electrónico basadas en el uso de dispositivos Beacon, válido en móviles Android.

Para conseguir este objetivo principal se tendrán en cuenta los siguientes objetivos secundarios:

- **Estructura común:** Diseño e implementación reutilizable de un esqueleto común para aplicaciones que requieran el uso de dispositivos Beacon, identificando los módulos que pueden ser usados por distintas aplicaciones, independientemente del objetivo concreto para que el que se desarrolle cada una de ellas. Por un lado, identificar todos los componentes implementados que puedan ser reutilizables y por otro lado, realizar el diseño y desarrollo de los componentes de la aplicación lo más independientes posibles.

- **Aplicación de prueba:** Desarrollo de una aplicación concreta que sirva de prueba de concepto de la estructura desarrollada anteriormente. En esta aplicación existirán uno o más dispositivos Beacons mediante los cuales será posible detectar la

presencia del dispositivo del cliente (el servicio de detección estará activo en segundo plano) y se le mandará una notificación/servicio/oferta personalizada.

El desarrollo de la aplicación de prueba será en Android y la base de datos será de Google Datastore para tener todo el servidor en la nube.

- **Uso de patrones de diseño**: Los patrones de diseño son muy necesarios tanto para mejorar la modularidad de los sistemas software, como para incrementar la reutilización de los módulos que lo forman. Este proyecto usará el patrón de diseño arquitectónico MVC, también llamada Modelo-Vista-Controlador. Usando este patrón, se separa el modelo de la aplicación de su interfaz gráfica, lo que puede permitir distintas interfaces para un mismo modelo de aplicación.

1.3 Estructura de la memoria

La estructura de esta memoria consta de varios capítulos, los cuales son:

- **Capítulo 1: Introducción.** Es donde nos encontramos actualmente y donde comienza la memoria, se trata objetivos del proyecto, además de la estructuración.
- **Capítulo 2: Tecnologías utilizadas**. En este fragmento se van a explicar el entorno tecnológico del plan, herramientas software y hardware empleado para conseguir con éxito este proyecto.
- **Capítulo 3: Requisitos y diseño de la aplicación.** En este capítulo se detalla una de las partes más importantes a la hora de realizar un proyecto como es su diseño. Se mostrarán distintos diseños pasando por sus respectivas iteraciones.
- **Capítulo 4: Implementación en Android.** Se especifica como se ha creado los distintos métodos de la estructura creada, además de la explicación de cómo usarla en una aplicación de prueba.
- **Capítulo 5: Implementación Servidor.** En este capítulo se tratarán las partes más importante en la parte del servidor, así como la creación de la API REST, sus servicios, y su posterior subida a la nube de Google Datastore.
- **Capítulo 6: Conclusiones y trabajo futuro**. En este capítulo se exponen las conclusiones obtenidas realizando el proyecto, así como las posibles líneas de trabajo futuro, es decir, distintas mejoras de nuestra estructura.
- **Referencias.**
- **Manual de instalación.**

2

Tecnologías Utilizadas

En este capítulo se van a explicar las distintas herramientas utilizadas durante el desarrollo de este proyecto, tanto software como hardware.

2.1 Tecnología Beacon

2.1.1 Descripción

Un Beacon es un dispositivo electrónico cuyo propósito es alertar a otros dispositivos móviles cercanos de su presencia mediante ondas de radio que usan el protocolo Bluetooth 4.0, también comúnmente llamado BLE [1]. Este dispositivo permite ubicar los dispositivos móviles cercanos con precisión, su tecnología está basada en la *difusión de mensajes* (*broadcasting* en inglés), es decir, un emisor envía información a todos los receptores (en este caso serían los dispositivos capaces de usar BLE) de la red. Estos reciben la información, la procesan, y existen infinidad de posibilidades a la hora de elegir una funcionalidad. A continuación, vemos en la figura 2 un modelo de *beacon*.

Figura 2: Imagen de un beacon de la marca FeasyBeacon. (Imagen tomada de la web oficial de FeasyCom - https://www.feasycom.com/product-254.html)

2.1.2 Protocolos utilizados.

Existen distintos protocolos a la hora de dar formato a los datos que envía el *beacon* a los usuarios, de este modo es más fácil para los programadores a la hora de elegir uno u otros. Hemos usado los 3 más frecuentes usados en el sector tecnológico, Eddystone (creado por Google), iBeacon (creado por Apple), Altbeacon (creado por Radius Networks). Vamos a describirlos a continuación.

Eddystone

Eddystone es un protocolo de código abierto desarrollado por Google, con soporte en iOS y Android. Define el formato de los mensajes BLE enviados por parte de un *beacon*, ofrece distintos tipos de formato, además de una arquitectura flexible que le permite evolucionar así como la incluso de nuevos parámetros. [2]

No se va a entrar en detalles sobre especificaciones, ya que no es el objetivo del proyecto. Se puede enviar uno de los siguientes paquetes gracias a este protocolo[3]:

- *Eddystone-UUID*: contiene un identificador de un *beacon*. El beacon emite un identificador único con 16 bytes que se encuentra separado en dos partes, 10 bytes pertenecen al *Namespace* y 6 bites pertenecen al *Instance* (ambos configurables por parte del desarrollador), lo que permite al desarrollador coordinar varios grupos de *beacons.* Usamos el *Namespace* para para definir los *beacon* a nivel grupal y el valor *instance* para definir el beacon individualmente.

- *Eddystone-URL*: contiene una URL que ocupa el mínimo espacio posible en los paquetes que manda el *beacon.*

- *Eddystone-TLM*: es emitido junto con los dos anteriores y contiene el estado de salud de un *beacon*, en este caso, el nivel de batería, u otra información como la temperatura del aparato o número de paquetes enviados.

- *Eddystone-EID*: contiene un identificador encriptado que cambia periódicamente. Creado con el objetivo de mejorar la privacidad entre el usuario y el *beacon*.

iBeacon

iBeacon desarrollado por Apple, fue el protocolo que introdujo por primera la tecnología BLE mundialmente y define 3 parámetros[4]:

- *UUID*: identificado un grupo.

- *Major*: identifica un subgrupo de *beacons* dentro de un grupo más grande.

- *Minor*: identifica un *beacon* especifico.

Comparándolo con *Eddystone*, incluye menos parámetros centrándose solo en el UUID, también permite definir distintas regiones, pero en vez de usar *Namespace* e *Instance* se usan *Major* y *Minor*. En la figura 3, se explican los distintos parámetros mencionados anteriormente;

Field	Size	Description
UUID	16 bytes	Application developers should define a UUID specific to their app and deployment use case.
Major	2 bytes	Further specifies a specific iBeacon and use case. For example, this could define a sub-region within a larger region defined by the UUID.
Minor	2 bytes	Allows further subdivision of region or use case, specified by the application developer.

Figura 3. Parámetros del protocolo iBeacon, creado por Apple [4].

Altbeacon

Este protocolo fue desarrollado por Radius Networks, fue una iniciativa para mejorar el protocolo cerrado de iBeacon, ofreciendo las mismas funcionalidades pero siendo capaz de entregar más información en cada mensaje emitido. Desarrollaron una librería la cual usaremos para configurar la detección de los beacons.

2.1.3 Simulador Beacon

Para evitar la compra de aparatos costosos para un beacon, existe una alternativa bastante buena como son los simuladores *Beacon*, la cual, un dispositivo que contenga la tecnología BLE puede simular a partir de apps un dispositivo *Beacon*. Existen aplicaciones que detectan *beacons* y otras que actúan como beacons y puedes cambiar los parámetros concretos de éstos.

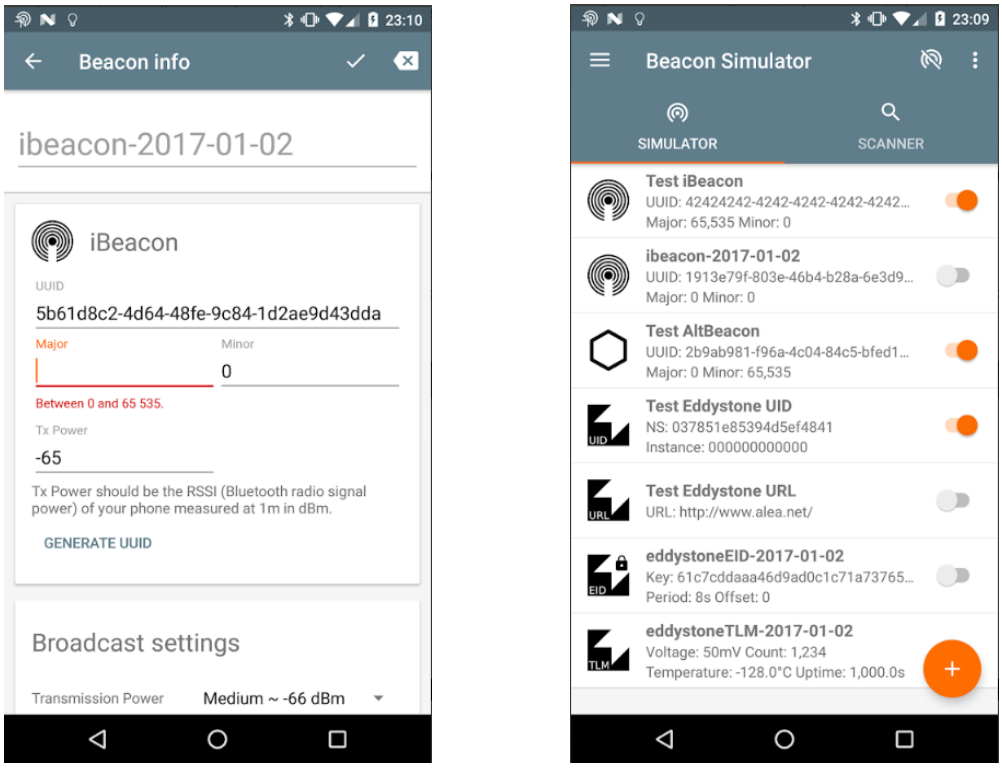

Figura 4. Capturas de una app que ofrece simulador y escáner de Beacon a través de un disp. Android.

2.2 Android

Android es un sistema operativo fundado por la empresa *Android INC*, más tarde fue comprada por Google, el cual continua desarrollando este sistema operativo completo libre y de código abierto. Por esto, presenta multitud de ventajas en cuanto a costes tecnológicos se refiere. El núcleo del sistema está basado en Linux que a su vez usa una máquina virtual *Dalvik* (primera versión) o ART(versión más reciente), siendo ambas compatibles.

Hemos usado para la aplicación prototipo del cliente, el IDE oficial para el desarrollo en Android, el llamado Android Studio. Este IDE cuenta con varias características que facilitan el desarrollo, compilación y ejecución de aplicaciones Android. A continuación mencionamos algunas de las características de este IDE [5]:

- Existe un emulador de dispositivos Android de distintas versiones para ejecutar las aplicaciones que son desarrolladas.
- Existe posibilidad de hacer control de versión mediante la plataforma *Github*.
- El compilado está basado en *Gradle*. Este *Gradle* sustituye los scripts de compilación basados en XML por el lenguaje DSL [6].
- Tiene una herramienta llamada *Lint* para detectar problemas.

Para nuestra aplicación prototipo hemos creado un proyecto, los archivos se organizan en módulos de manera que cada uno tiene una determinada función y compilado individual [7]. Cada módulo debe contener las siguientes carpetas[5]:

- *Manifest*: Ubicación del archivo androidManifest.xml, donde contiene toda la información clave de la ejecución de la app, contiene las actividades, servicios, permisos, entre otros.
- *Java*: Aquí se encuentra todos los archivos con el código fuente.
- *Res*: Distintos archivos que la aplicación usa como recursos como nombres, estilo, etc.

En este proyecto se ha usado un smartphone Samsung Galaxy A5 con un sistema operativo Android, para ejecutar la aplicación prototipo. No todas las versiones soportan la tecnología BLE, la versión mínima del sistema operativo es 4.3 (API 18).

2.3 Librería Retrofit

Retrofit es una librería de código libre que se encarga de hacer llamadas REST a un servidor. Sin esta librería tendríamos que usar una llamada con HttpClient y luego usar *json.get* para construir los objetos, pero con *Retrofit* nos ahorramos todo eso. [8]

Junto a *Retrofit* se usa un serializador soportado en su página, en nuestro caso usamos gson.

2.4 Google Gson

Combinado con Retrofit usamos una librería de Java para transformar objetos Java a su representación en formato JSON (JavaScript Object Notation) y a su vez de JSON a objetos Java. Esta librería se llama Gson. Lo podemos encontrar en la url: https://github.com/google/gson.

Gracias a esta librería podemos convertir instancias de Set, Map, List.

Con esta línea convertimos cualquier objeto a JSON.

```
new Gson().toJson(object);
```

En cambio para recuperarlo tenemos dos posibilidades, la primera puede ser una clase:

```
Agenda agenda = new Gson().fromJson(jsonString, Agenda.class);
```

O en el caso que sea una implementación de una interfaz, ya sea una lista, un mapa o un conjunto;

```
List<Nombre> nombres = new Gson().fromJson(jsonString, new TypeToken<List<Nombre>>());
```

Todo esto lo hace la librería Retrofit en nuestro lugar, pasar un objeto del servidor al cliente.

2.5 Google Datastore

A la hora de hacer aplicaciones online es necesario guardar la información, por la tanto, necesitamos una base de datos donde poder almacenar, modificar y recuperar información de manera rápida y eficaz. Hemos utilizado para la aplicación prototipo, Google Datastore por varias razones. Es una base de datos NoSQL (no relacional) altamente escalable, entre otras cosas, se encarga automáticamente de la fragmentación y replicación. Tiene una interfaz Restful donde se puede desplegar fácilmente y acceder a los datos, además, tiene un motor de consultas muy variado que permite buscar datos en varias propiedades y ordenarlos según la necesidad.

Google Datastore es un servicio ofrecido por Google en la plataforma de Google Cloud. Google Cloud Datastore permite a los usuarios crear base de datos en modo nativo o modo Datastore. El modo nativo está diseñado para móviles y aplicaciones web mientras que el modo Datastore está diseñado para nuevos proyectos de servidor. [9]

Tiene distintos precios pero en nuestro caso la versión gratuita de un año es más que suficiente para el proyecto.

2.6 Python con Flask

Python es un lenguaje de programación interpretado, dinámico y multiplataforma. Se centra mucho en la legibilidad del código, esto hace que sea más fácil su depuración y favorece la productividad. Su característica Open Source (código abierto) junto a su facilidad de aprendizaje hace que sea un potente lenguaje para muchos. [10]

Este lenguaje fue creado por Guido Rossum en 1991 en los Países Bajos como sucesor del lenguaje de programación *ABC*[11]. Python es un lenguaje de programación multiparadigma, lo que permite varios estilos como por ejemplo la programación imperativa u orientada a objetos. Usa un tipado dinámico como habíamos mencionado anteriormente y conteo de referencias para la administración de la memoria. Una de las características más importante es la resolución dinámica de nombre, es decir, aquello que enlaza un método y un nombre de una variable durante la ejecución del programa.

En *Python* existe una gran filosofía, donde algunos la comparan con la filosofía de *Unix*. Los códigos que sigan los principios de esta filosofía se dicen que son códigos "pythonicos" , estos principios fueron desarrollados por Tim Peters en El Zen de Python.

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Figura 5: Principios del libro El Zen de Python escrito por Tim Peters.[12]

Además de Python vamos a usar un Framework escrito en Python para desarrollar aplicaciones WEB llamado *Flask* bajo el patrón que seleccionamos; Modelo-Vista-Controlador.

Flask es una herramienta que nos dan un esquema de trabajo y distintas funciones que nos facilita y nos abstrae de la construcción de páginas web dinámicas. Entre las ventajas que encontramos en *Flask son*;

- Es Open Source como Python, bajo la licencia BSD (Berkeley Software Distribution).
- Tiene un depurador integrado, sirve de mucha ayuda para pruebas cuando tenemos algún error en el código.
- Incluye un servidor web para probar las aplicaciones. No es necesaria ninguna infraestructura con servidor web, es una facilidad a la hora de ir viendo los resultados que se van obteniendo.
- Tiene un buen manejo de rutas, el controlador recibe todas las peticiones que hace los clientes y tiene que determinar que ruta está accediendo el cliente para ejecutar el código necesario.
- Es compatible con python3.
- Posibilidad de usar sesiones.
- Es compatible también con wsgi, este es un protocolo que utiliza los servidores web para servir las pagina web escritos en python (No lo usaremos).

Existen varias extensiones que complementan *Flask* a la hora de hacer aplicaciones web, por ejemplo; *flask-login*, nos permite autenticación de usuario y contraseña, *flask-script*, nos permite tener un comando de la línea de comando para manejar la aplicación o *flask-Bootstrap*, donde nos permite cambiar el estilo para las páginas web, entre otros. [13] Actualmente usamos para nuestro proyecto la versión 3.7 para crear la API REST.

3

Requisitos y diseño

En este capítulo se detallan datos muy importante de las distintas iteraciones que se proponen para hacer el esqueleto de la estructura común, además del diseño de una aplicación simple a partir de esta estructura y su interfaz gráfica.

3.1 Especificaciones

Vamos a realizar un proyecto donde se pretende hacer un caso de uso de la tecnología de proximidad de tipo *beacon.* Este caso de uso está orientado al comercio electrónico (e-commerce, en inglés), por tanto centrándonos en la atención que se le puede dar al usuario cerca de una tienda o centro comercial.

A continuación, en la siguiente figura se muestran los dispositivos que son necesarios para el proyecto. Un dispositivo Beacon, un smartphone, conexión a internet y un servidor con nuestra base de datos.

Figura 6. Componentes que participan en el proyecto.

Al final del proyecto se deben haber cumplido los siguientes objetivos:

- Configurar uno o varios dispositivos de tipo *beacon* con los parámetros necesarios para interactuar con otros dispositivos cercanos.

- Desarrollar una aplicación capaz de detectar los dispositivos de tipo *beacon* cercanos y distintas características suyas como distancia o distribuidor.
- Desarrollar un cliente, capaz de hacer una llamada REST al detectar un *beacon*.
- Desarrollar un servidor capaz de responder a las peticiones del cliente.
- Realizar una API REST con todas las llamadas posibles que pueda hacer el cliente.
- Trabajar con la base de datos para interactuar con la información recibida y comunicársela al servidor. La base de datos podrá ser de cualquier tipo no relacional.

Todos los objetivos anteriores deben ir encaminados siempre a conseguir nuestro objetivo principal, que es la definición de una estructura común que pueda ser reutilizada en el desarrollo de aplicaciones similares. Es decir, la aplicación desarrollada nos va a permitir identificar todas esas características comunes. Realizaremos una estructura que pueda ser usada por cualquier programador para sistemas operativos Android, donde cada programador pueda aportar su servidor y base de datos independiente de la estructura o de lo que nosotros usemos en nuestro prototipo. Por lo que, en los siguientes capítulos veremos cómo realizamos una estructura común y a partir de ella la realización de una aplicación simple.

Tras estos objetivos mencionados, el propósito general es usar tecnologías aprendidas durante toda la titulación además de alguna nueva, aplicando todos los posibles conocimientos aprendidos.

3.2 Requisitos funcionales de la estructura común

Los requisitos describen la funcionalidad que ofrece un sistema y sus restricciones a la hora de la implementación. Es una declaración abstracta de alto nivel de un servicio que el sistema debe proporcionar. Ahora vamos a explicar los distintos requisitos necesarios para la estructura común.

- El sistema debe detectar el *beacon* si pertenece a su rango de alcance.
- El sistema podrá leer información proporcionada por el *beacon*.
- El sistema debe permitir buscar distintos tipos de *beacons*.
- El sistema debe permitir filtrar los distintos *beacons* dentro de un mismo tipo.
- El sistema permite cambiar el tiempo de escaneo entre periodo.
- El sistema debe funcionar en segundo plano.

3.3 Requisitos no funcionales de la estructura común

A la hora de realizar un proyecto con estas características existe un mínimo de reglas que se deben cumplir para desarrollar una aplicación con éxito, entre las cuales son:

- Esta estructura esta creada a partir de Android Studio, por tanto estaría limitada a dispositivos con sistema Android, mientras que la parte del servidor es completamente multiplataforma.
- El sistema Android deberá tener una versión mayor a 4.3 (Jelly Bean) y tener la capacidad de trasmitir Bluetooth Low Energy (prácticamente todos los dispositivos implementan esta característica).

- Otro requisito no funcional para desarrollar una aplicación con estas características y permitir permisos de red, localización, etc. se debe tener un SDK (Software Development Kit) igual o mayor a la versión 29.
- El dispositivo debe tener acceso a internet para acceder a la funcionalidad del servicio completo.
- La seguridad entre la aplicación será Basic Auth (Autenticación de acceso básica). Es la forma más básica de autenticación disponible para aplicaciones web y solo usa codificación Base64. Al ser una comunicación privada cliente-servidor y no se accede a terceras partes, es suficiente para la aplicación prototipo.

3.4 Requisitos de la aplicación prototipo

3.4.1 Requisitos funcionales

Los requisitos funcionales son aquellos que describen lo que debe hacer un sistema, donde se detalla cómo se debe comportar ante los distintos escenarios.

- **REQ. F. 1.** El usuario debe iniciar sesión en la aplicación.
- **REQ. F. 2.** El usuario puede localizar tiendas.
- **REQ. F. 3.** El usuario puede buscar sus ofertas.
- **REQ. F. 4.** El sistema puede mandar notificaciones al usuario.
- **REQ. F. 5.** El usuario puede ver las citas que tiene.
- **REQ. F. 6.** El usuario puede ver sus imágenes.
- **REQ. F. 7.** El usuario puede ver sus mapas.
- **REQ. F. 8.** El usuario puede ver sus productos personalizados en oferta.

3.4.2 Requisitos no funcionales

Estos requisitos son aquellos que definen el "como" del sistema. Describen distintas restricciones que afectan a la funcionalidad de la aplicación o sistema. Suelen estar relacionados con el lenguaje de programación, calidad del rendimiento, seguridad, mantenimiento entre otros.

- **REQ. NF. 1.** La aplicación cliente debe ser desarrollada en Android.
- **REQ. NF. 2.** Interfaz amigable, fácil de usar.
- **REQ. NF. 3.** El servidor será subido a la nube en Google Cloud.
- **REQ. NF. 4.** Sistema Android debe tener una versión 4.3 o posterior.
- **REQ. NF. 5.** El SDF debe ser una versión igual o posterior a 29.
- **REQ. NF. 6.** El sistema debe ser accesible a Internet y localización.
- **REQ. NF. 7.** La seguridad entre aplicación y servidor debe ser Basic Auth.
- **REQ. NF. 8.** El sistema debe usar el patrón Modelo-Vista-Controlador.

3.5 Casos de usos

En la figura 7 podemos ver el caso de uso simple del usuario interaccionando con la aplicación de prueba, pero lo realmente importante está en la figura 8, que representa

un caso de uso de lo que realiza en segundo plano nuestro sistema sin la interacción directa con el usuario. Podemos ver, que puede distinguir señales de distintos tipos de beacons, filtrar esa información, y realizar llamadas al servidor.

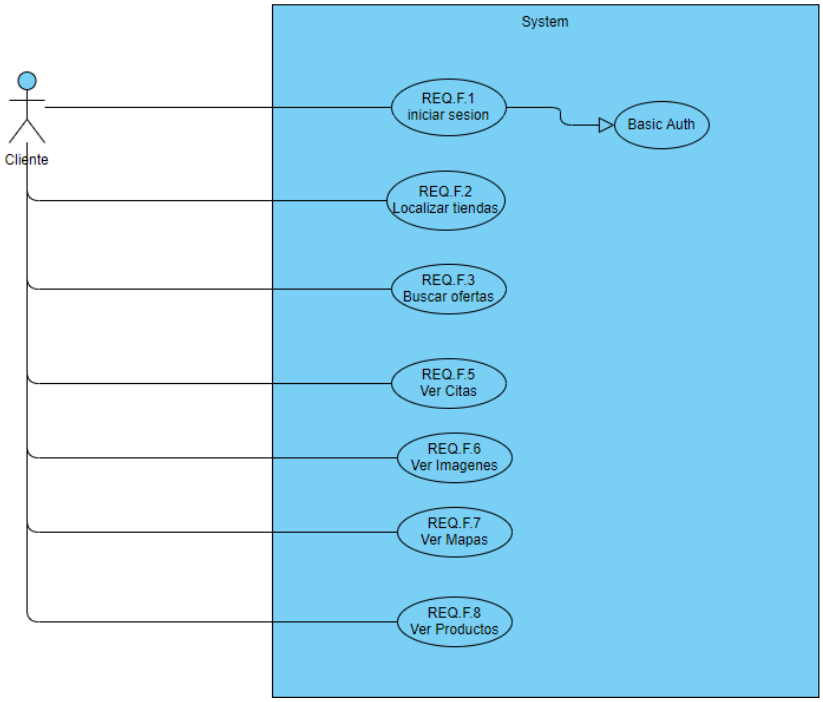

Figura 7. Diagrama de caso de uso desde la perspectiva del cliente.

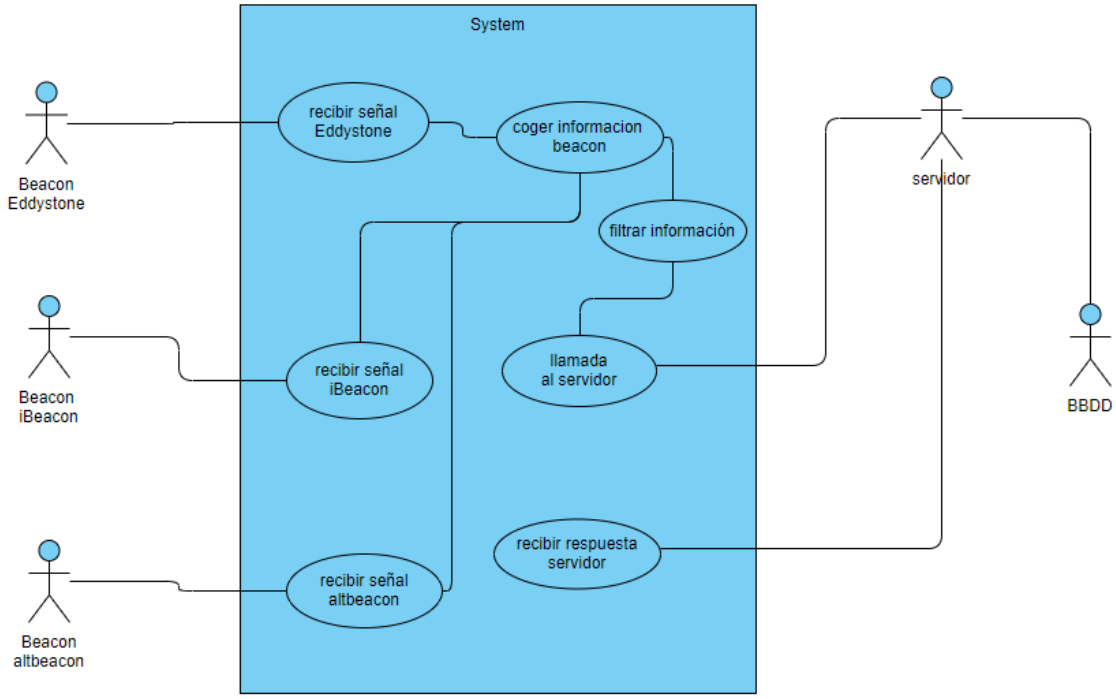

Figura 8. Diagrama de caso de uso de la estructura común.

3.6 Patrón de diseño Modelo-Vista-Controlador

Este patrón fue desarrollado por Trygve Reenskaug en Palo Alto Research Center de Xerox Parc en 1979. Fue uno de los primeros trabajos en describir e implementar las aplicaciones software en términos de sus diferentes funciones.

El objetivo de este patrón es clasificar la información (modelo), la lógica del sistema (controlador) y la interfaz (vista) que se le ofrece al usuario. Esto sirve para reducir la complejidad en el diseño arquitectónico e incrementar la flexibilidad y mantenimiento del código. En otras palabras, este patrón separa los datos, la interfaz y la lógica d negocio.

El **modelo** es el componente encargado de manipular, gestionar y actualizar datos, por ejemplo, si se utiliza una base de datos, aquí es donde se realizan las consultas, búsquedas, filtros etc. Las peticiones de acceso o manipulación de la información llegan a la vista a través del modelo.

La **vista** es simplemente el componente encargado de mostrar al usuario las pantallas, paginas, etc. Es el componente encargado del frontend.

El **controlador** es el componente intermediario entre el modelo y la vista, representa la lógica de la aplicación.

En **Android** es muy fácil distinguir estos tres componentes, la vista, serían los propios layouts, el modelo, son las clases que representan los datos y el controlador, son las actividades de la aplicación.

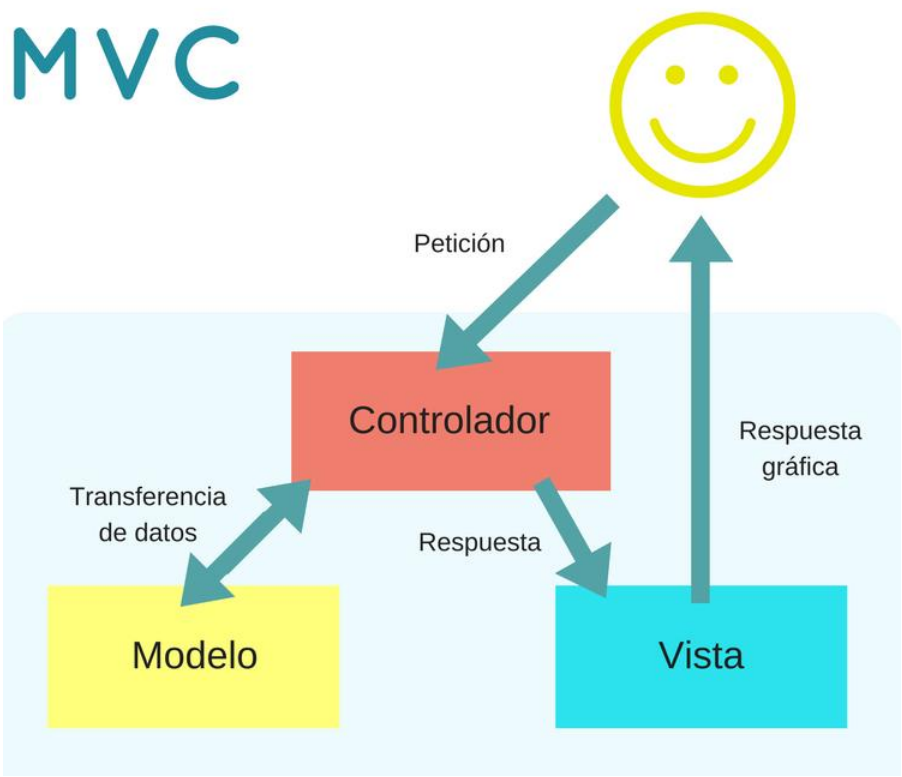

Figura 9. Los distintos componentes del patrón MVC. (Imagen tomada de la web https://codingornot.com/mvc-modelo-vista-controlador-que-es-y-para-que-sirve)

3.7 Prototipado de la aplicación de prueba

Hemos realizado una aplicación para probar la estructura común que explicaremos más adelante, en el siguiente capítulo, como ha sido creada. En este apartado veremos solo la interfaz gráfica de la aplicación de prueba y lo que nos muestra nuestra API REST creada. La aplicación ha sido creada en Android Studio con una base de datos de Google Cloud Datastore.

Esta aplicación de muestra ayuda para que se vea como es la funcionalidad del proyecto. A continuación se muestra el diseño de todas las pantallas de la aplicación:

3.7.1 Notificación

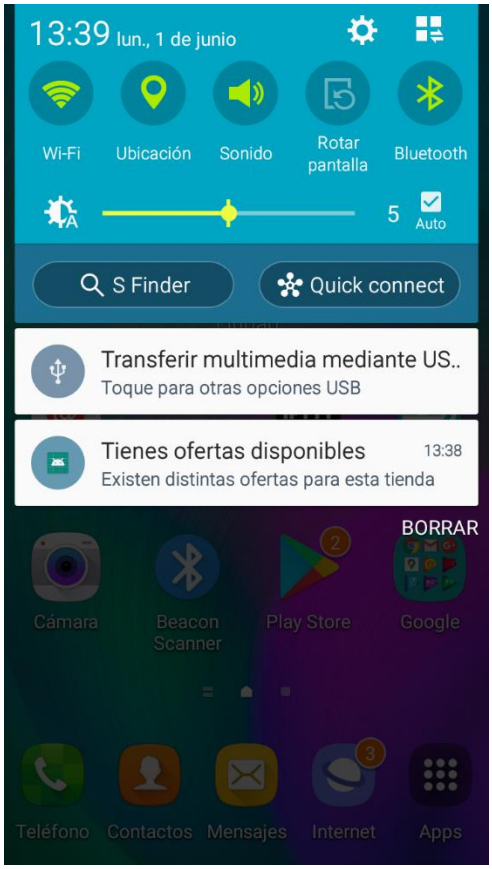

Figura 10. Imagen del centro de notificaciones del sistema Android donde se ejecuta la aplicación de prueba.

Vemos en esta captura de pantalla, una notificación que se ejecuta cuando detecta un beacon de nuestra región a nuestro alcance. Al hacer clic en esta notificación nos llevará a la pantalla inicio de nuestra aplicación. En el siguiente capítulo se explicará su funcionamiento.

3.7.2 Pantalla inicio

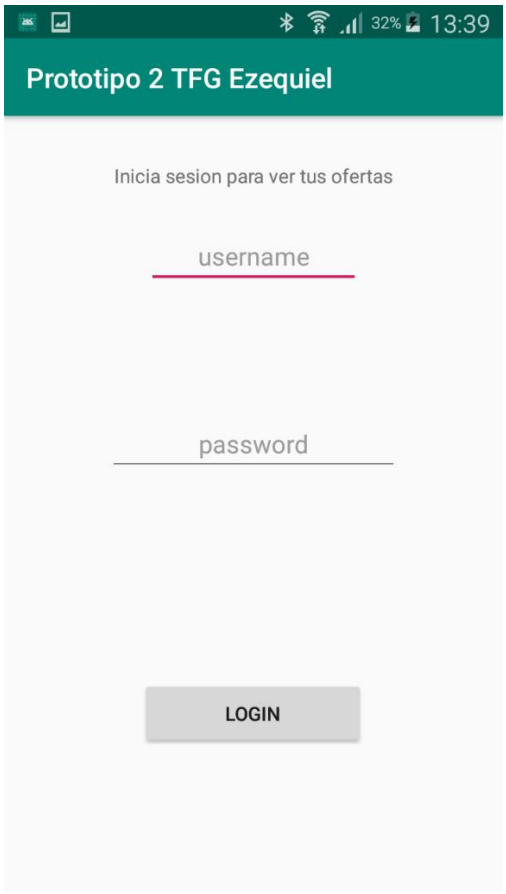

Figura 11: Captura de pantalla de la actividad inicial de la aplicación de prueba.

Esta es nuestra primera pantalla, donde tenemos que iniciar sesión con un usuario, y una contraseña ya creados previamente en la base de datos. Al pulsar sobre el botón "LOGIN" nos dirige a la siguiente actividad. Esta pantalla no pertenece a la estructura común ya que el programador decidirá que pantalla de inicio realizar.

3.7.3 Segunda actividad

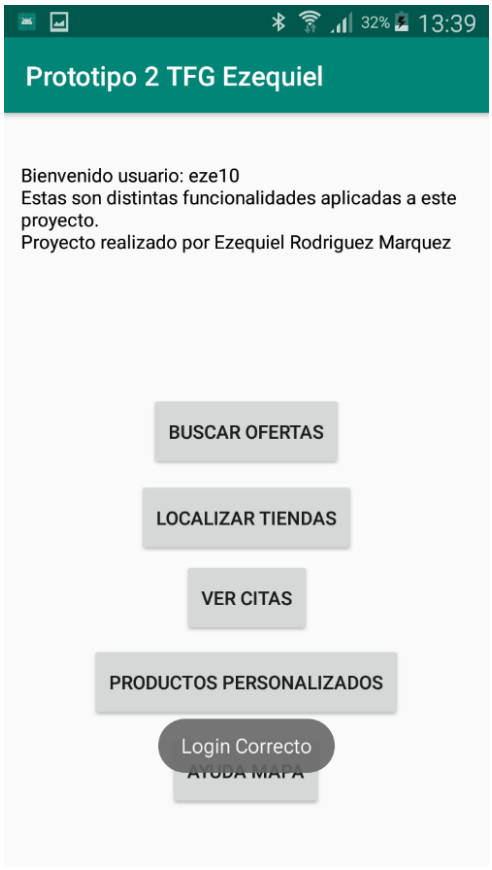

Figura 12: Captura de pantalla de la segunda actividad de nuestra aplicación de prueba.

Esta actividad nos muestra un TextView en la parte superior indicando quien ha iniciado sesión. En la parte inferior vemos cinco botones, cada uno nos redirigirá a una actividad distinta:

- El botón **"BUSCAR OFERTAS"** nos dirige a la actividad llamada *"MonitoringActivity"*.

- El botón **"LOCALIZAR TIENDAS"** nos dirige a la actividad llamada *"RangingActivity"*.

- El botón **"VER CITAS"** nos dirige a la actividad llamada *"CitasActivity".*

- El botón **PRODUCTOS PERSONALIZADOS"** nos dirige a la actividad llamada *"ProductsActivity".*

- El botón "**AYUDA MAPA"** nos dirige a la actividad llamada *"ImagenActivity".*

Cada uno de estos botones nos lleva a una funcionalidad que se podría aplicar usando el esqueleto creado en este proyecto.

3.7.4 MonitoringActivity

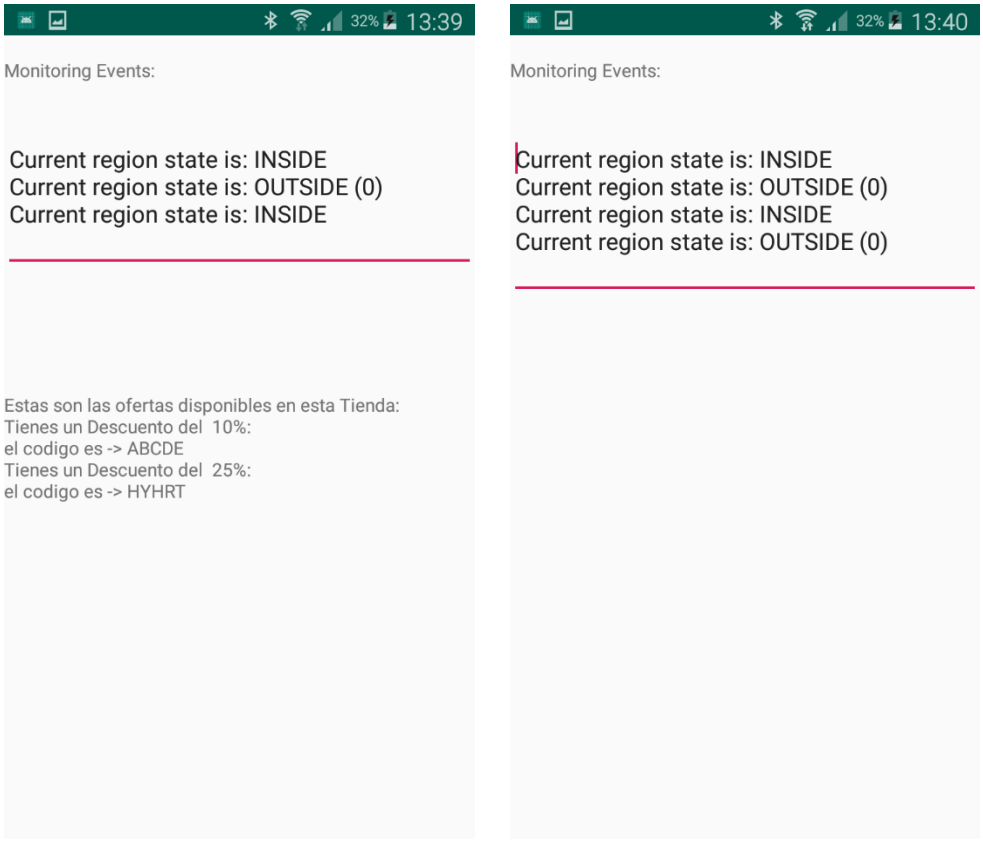

Figura 13. Captura de pantalla de la actividad "*MonitoringActivity"*, a la izquierda cuando ha detectado al *beacon*, a la derecha cuando está fuera del rango del *beacon.*

Esta actividad nos indica las ofertas existentes para el usuario que ha iniciado sesión en la tienda. Cuando desaparece del rango del *beacon,* las ofertas desaparecerían y solo volverían a aparecer cuando estuviera dentro del rango otra vez. Si fuera una app de un centro comercial se podrían ver las distintas ofertas de cada tienda al acercarse al rango del beacon sin tener la necesidad de tener varias apps instaladas.

La parte superior nos indica si estamos o no, dentro del rango del beacon, indicando *"Current region state is: INSIDE*", cuando estamos dentro, y *"Current region state is: OUTSIDE(0)"* cuando estamos fuera de ese rango. En la parte inferior vemos un TextView donde nos muestra las ofertas.

3.7.5 RangingActivity

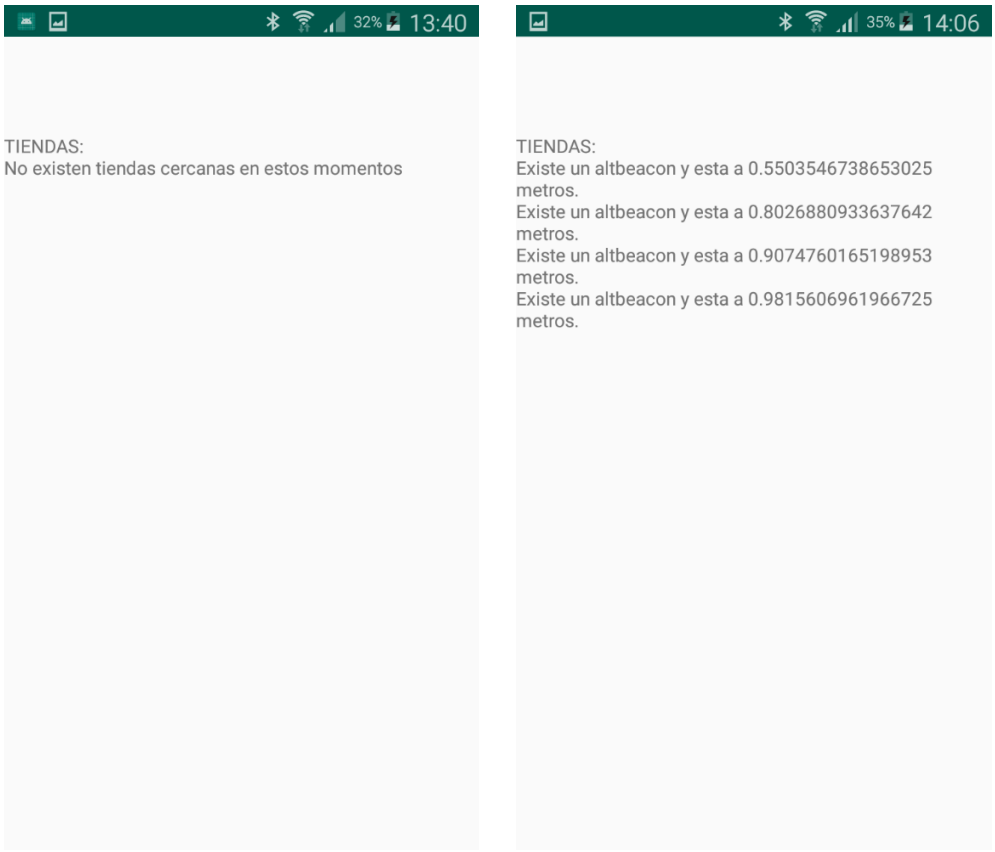

Figura 14: Captura de pantalla de la actividad "*RangingActivity*".

Esta actividad nos muestra los *beacons* a nuestro alcance, indicando la distancia a la que nos encontramos de ellos. Una funcionalidad sería indicar donde nos encontramos en un centro comercial y donde están las tiendas. Otra posibilidad es, realizar una acción cuando estamos a menos de un número de metros, ya que tenemos un valor que nos indica la distancia y podemos usarlo.

3.7.6 CitasActivity

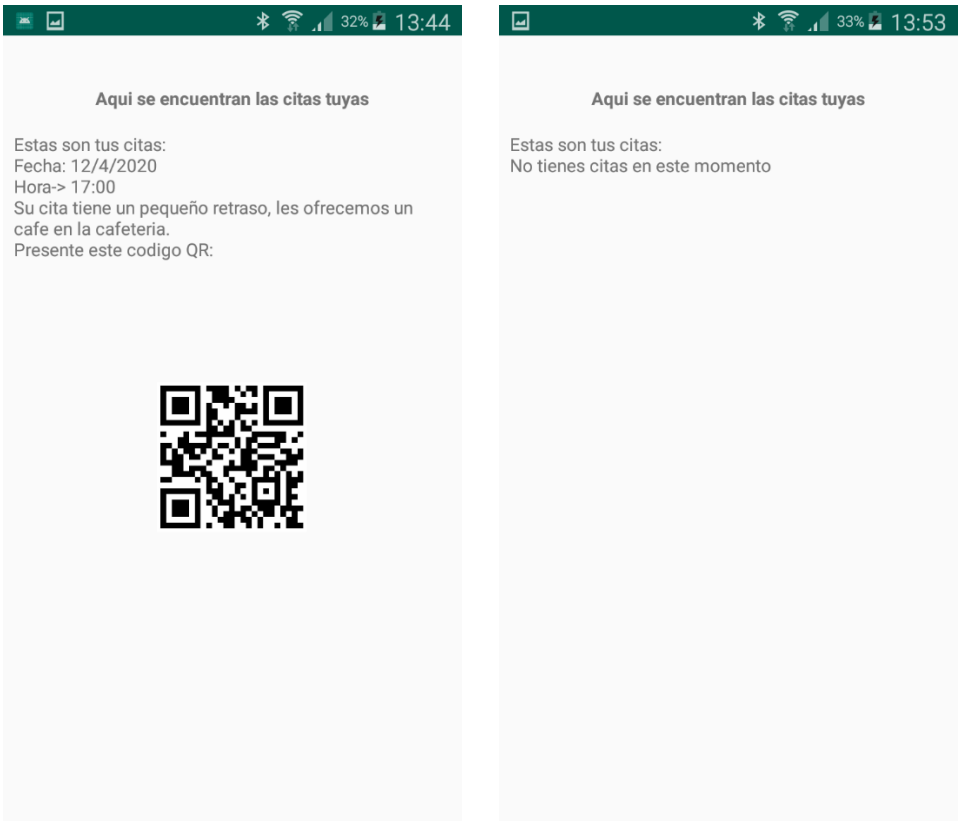

Figura 15. Captura de pantalla de la actividad "*CitasActivity*". En la izquierda estamos dentro del rango y en la derecha estamos fuera del rango.

En esta actividad, simulamos un negocio con un sistema de citas. Cuando llegamos a la tienda, podemos a abrir la app, y vemos si nuestra cita va con retraso, como posibilidad, si la cita en cuestión lleva un retraso nos avisa y nos ofrece un café mientras esperamos. Esto es una posibilidad entre tantas que existen, otra opción sería, dar la posibilidad de aplazar la cita, ofrecer algún descuento o en el caso de una tienda donde se recojan pedidos, enseñarle a donde tiene que dirigirse para recoger el pedido, entre otras opciones.

3.7.7 ProductsActivity

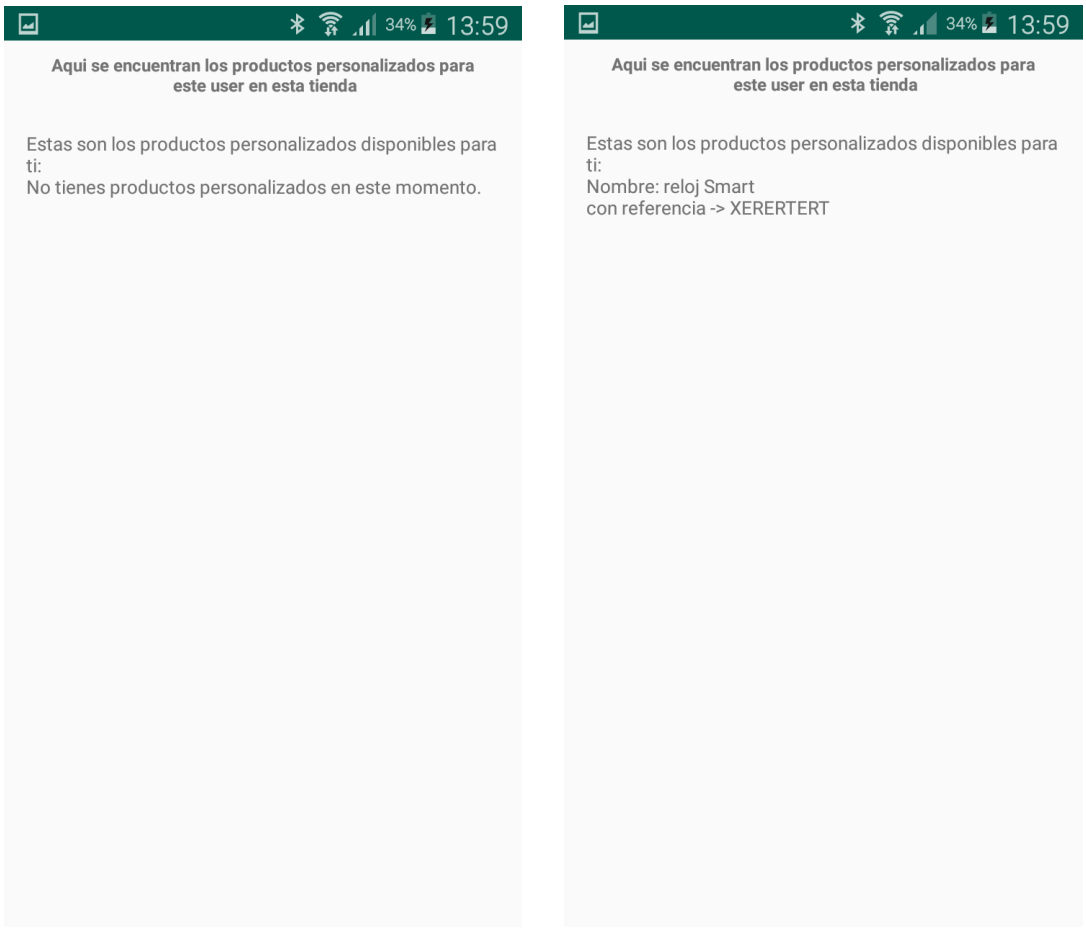

Figura 16. Captura de pantalla de la actividad *"ProductsActivity".* En la izquierda estamos fuera del rango del *beacon* y en la derecha estamos dentro del rango del *beacon.*

En esta actividad, podemos ver, cuando nos acercamos a un comercio, la aplicación nos ofrece unos productos personalizados, ya sea porque llevamos tiempo buscando ese producto. Como mejora se podría añadir una imagen del producto o más información del producto en cuestión. También se debería poner un precio al producto pero como esta aplicación es solo una prueba no hemos decidido ponerlo.

3.7.8 ImagenActivity

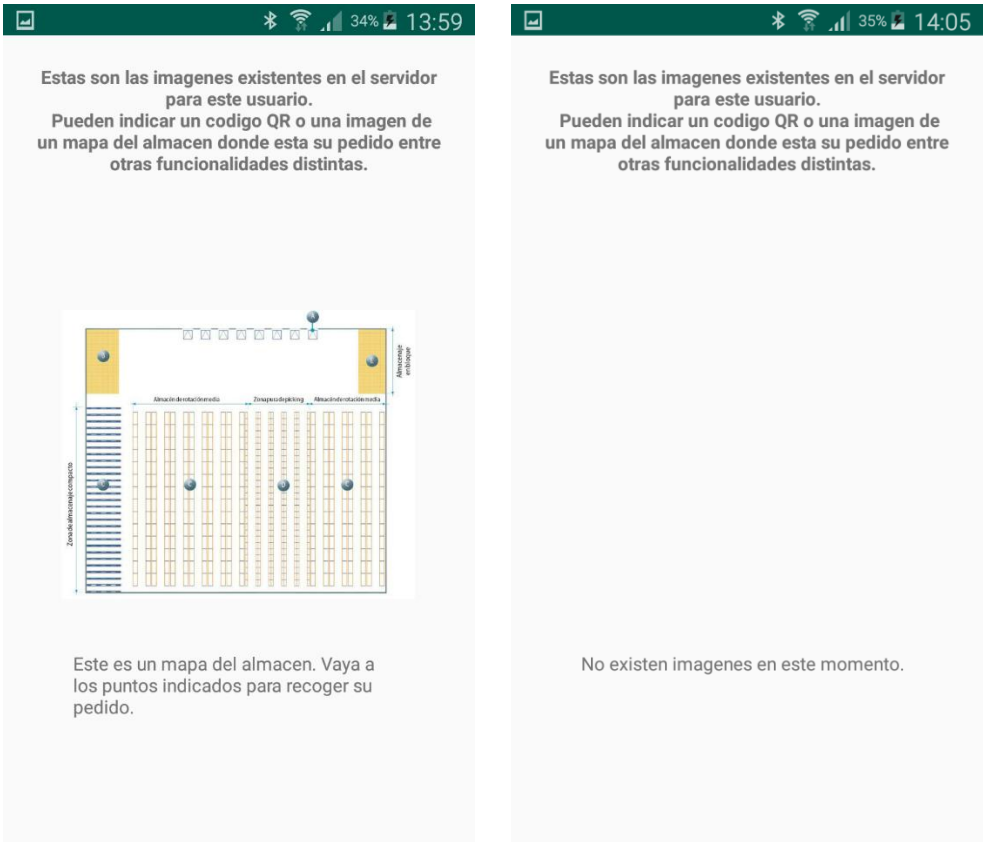

Figura 17. Captura de pantalla de la actividad "*ImagenActivity*". En la izquierda estamos dentro del rango del *beacon* y en la derecha estamos fuera del rango.

En esta actividad, nos encontramos por ejemplo en un almacén y la aplicación nos enseña un mapa donde está situado el producto que estamos buscando o que hemos seleccionado para comprar y ahora tenemos que recoger.

3.8 Comunicación Cliente-Servidor

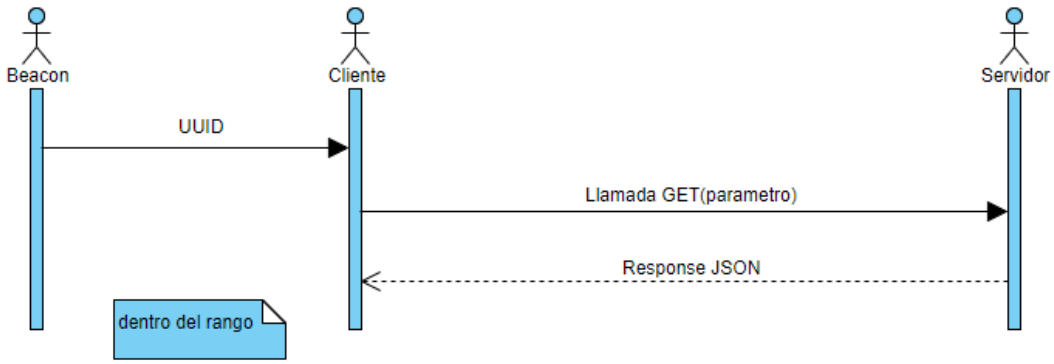

Figura 18. Diagrama de secuencia de la comunicación entre el Cliente-Servidor.

En la figura 18 podemos ver de manera simple la comunicación entre el cliente y servidor junto al *beacon* trasmisor. Podemos explicarlos en tres partes:

- En la primera parte, el beacon está continuamente trasmitiendo información a los receptores de *Bluetooth Low Energy*. En este caso recibimos el *UUID*, y nuestro smartphone recibe la señal.

- En la segunda parte, el cliente realiza una llamada al servidor a través de una URL, pasándole parámetros y en la cabecera una identificación (usuario y contraseña codificado en Base64), ya que entonces, sin seguridad, cualquier usuario podría ver la información con la URL solicitada. Esta seguridad es la más básica posible y suficiente al no tener terceros servicios.

- Por último, el servidor recibe la llamada del cliente, y devuelve en formato *JSON*, un archivo con el resultado dado.

3.9 Diseño Base de Datos

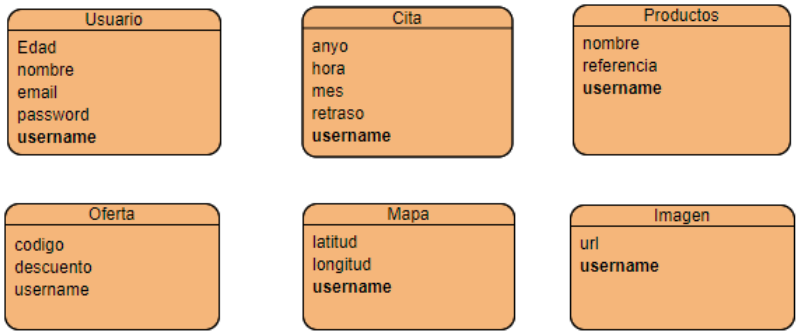

Figura 19. Diseño conceptual simple de la base de datos empleada en la aplicación de prueba.

Este es el simple diseño que he usado para mi aplicación. Una base de datos no relacional, donde tenemos entidades con la información. Uso un dato identificativo para todo, el *username*, ya que es único para todos los usuarios, además, Google Cloud Datastore asigna automáticamente un ID a cada instancia que se cre a.

La base de datos es una parte dependiente de cada proyecto y, por tanto, cada usuario puede aportar la suya.

3.10 Diagrama general de secuencia

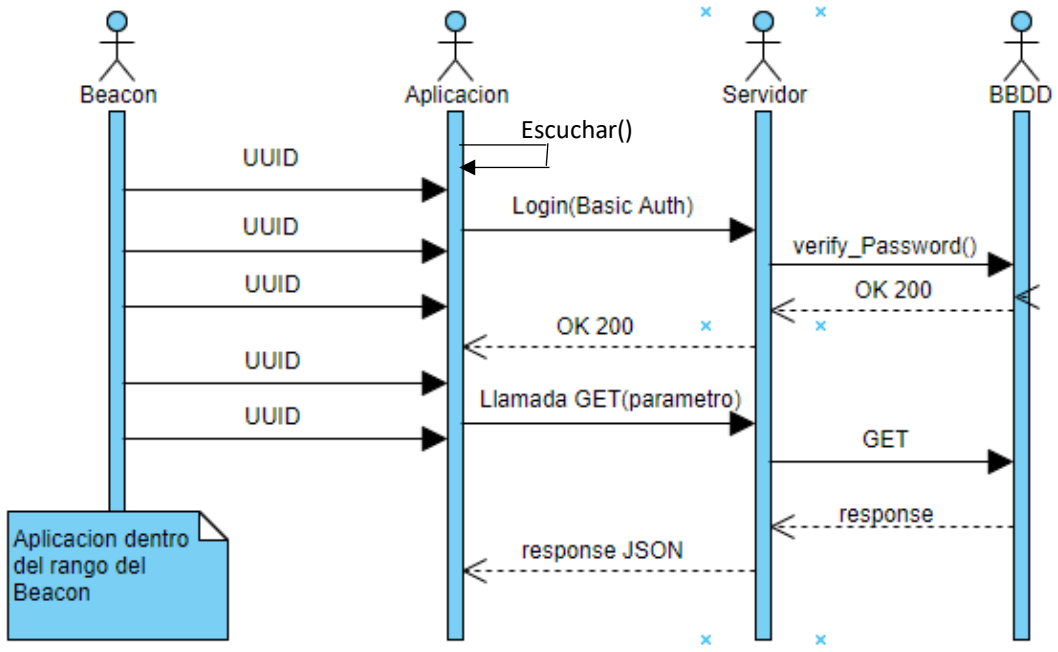

Figura 20. Diagrama de secuencia del proyecto.

En la figura 20, podemos observar un diagrama de secuencia donde se explica cómo interactúan los diferentes elementos del sistema creado. El diagrama empieza cuando la aplicación entra dentro del rango de trasmisión del *beacon,* pero antes la aplicación ha iniciado su función bluetooth y la detección de *beacons* (indicado con la función *escuchar()*). Ahora nuestro dispositivo está detectando los *beacons* y recibimos la información, ahora la aplicación nos pide una identificación que nos da seguridad entre la aplicación y el servidor explicado anteriormente. Una vez nos hemos identificados se hace una llamada al servidor.

El servidor accede a la base de datos cogiendo la información solicitada por el usuario, este transforma la información a formato JSON y se la trasmite a la aplicación.

Nuestra idea es crear una estructura reutilizable donde cada usuario puede crear un proyecto independientemente de la BBDD y del servidor.

4

Implementación en Android

En este episodio se especifica como se ha creado los distintos métodos de la estructura creada, además de la explicación de cómo usarla en una aplicación de prueba.

4.1 Ciclo de Vida en Android

En Android, solo existe una actividad activa en cada momento, por tanto, Android gestiona los distintos cambios entre primer plano y segundo plano de las aplicaciones y eso es lo que vamos a explicar en este apartado que se detalla a continuación. [18]

Una actividad tiene un ciclo de vida que está compuesto por estados y transiciones que definen el comportamiento de la actividad al pasar de un estado a otro.

Existen cuatro estados en los cuales puede estar una actividad. Son los siguientes:

- Running: Este es el estado que indica que la actividad está en primer plano y por tanto, es la actividad que el usuario está visualizando en ese mismo instante y con la que puede interactuar.

- Paused: La actividad en este estado estará visible pero no en primer plano sino en segundo plano (*background*, en íngles). Existe un riesgo que *Android* elimine esta actividad en casos de que haya memoria muy baja.

- Stopped: La actividad se encuentra en este estado cuando está totalmente oculta y existe otra actividad activa. Existe el mismo riesgo que el estado *Paused* de que la actividad sea eliminada en casos de memoria baja.

- **Shut down:** Una actividad se encontrará en este estado si no se encuentra en ninguno de los estados anteriores, ya sea porque haya sido eliminada o incluso porque no se haya iniciado.

En la Figura 21 podemos ver el ciclo de vida de una actividad por sus distintos estados, donde son los siguientes:

- **onCreate():** Este es el método inicial, cuando se inicia la actividad por primera vez y antes de que se muestre la interfaz de usuario, este método es llamado. Ya que este es el método que se encarga de configurar la interfaz gráfica que usará la aplicación mediante el método *setContentView().*

- **onResume():** Este método es llamado justo antes de entrar en el estado running, se llama después de la llamada a onStart() o cuando es recomenzada desde el estado *paused.*

- **onPause():** Lógicamente este método es llamado antes de entrar en el estado *paused*. Es el único método que se asegura que es llamado cuando la aplicación esta pausada, por lo que, puede ser utilizada para salvar los datos de la aplicación.

- **onDestroy():** Este método es llamado antes de pasar por el estado *shut down* pero no se asegura de que sea llamada, por ejemplo, si el sistema *Android* decide cerrarla por falta de memoria.

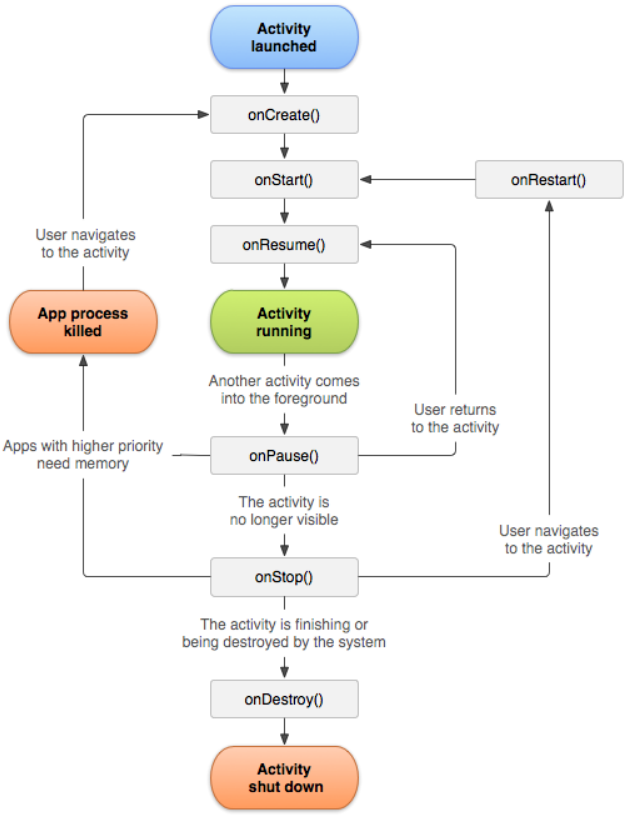

Figura 21. Ciclo de vida de una actividad. Imagen sacada de la URL https://developer.android.com/guide/components/activities/activity-lifecycle

4.2 Estructura reutilizable

Esta estructura está formada principalmente por una aplicación que controla dos actividades, una actividad encargada de monitorizar los beacons y otro de detectar la información que le dan los beacons, como por ejemplo la distancia. Entre otros elementos, se encuentra una interfaz y distintas entidades para realizar llamadas a un servidor externo.

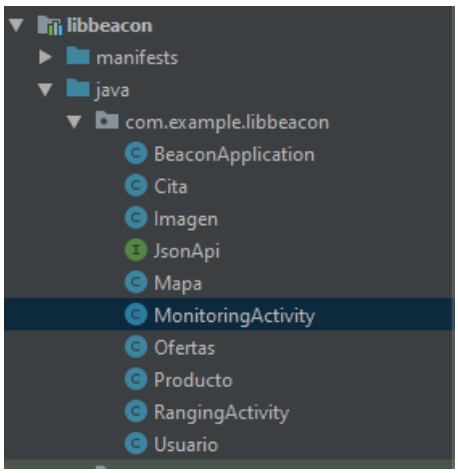

Figura 22. Elementos de la estructura reutilizable.

4.2.1 BeaconApplication

Es la clase Aplicación que dirige las actividades *MonitoringActivity* y *RangingActivity*, implementa también *BootstrapNotifier* que nos servirá para mandar aplicaciones. A continuación mostramos el método *onCreate* de nuestra clase aplicación.

```
    @Override
    public void onCreate() {
        super.onCreate();
        Log.d(TAG, "App started up");
        BeaconManager beaconManager =
BeaconManager.getInstanceForApplication(this);
        beaconManager.setDebug(true);
        // beaconManager.getBeaconParsers().add(new BeaconParser().
        //        setBeaconLayout("m:2-3=beac,i:4-19,i:20-21,i:22-
23,p:24-24,d:25-25"));
        Region region = new Region("boostrapRegion", null,null,null);
        regionBootstrap = new RegionBootstrap(this, region);
        beaconManager.setBackgroundBetweenScanPeriod(0);
        beaconManager.setBackgroundScanPeriod(1100);
        backgroundPowerSaver = new BackgroundPowerSaver(this);
    }
```

Este método inicializa un *BeaconManager* que se encarga de realizar la búsqueda de *beacons*, este ha sido creado en la librería *altbeacon*. Por defecto esta librería busca

solo *beacons* de tipo altBeacon. Tenemos la opción de cambiar la búsqueda a otros tipos de beacons ya explicados anteriormente, para ello, tenemos que añadir una línea más, la cual, esta comentada en el código anterior y es la siguiente:

```
beaconManager.getBeaconParsers().add(new BeaconParser().
setBeaconLayout("m:2-3=beac,i:4-19,i:20-21,i:22-23,p:24-24,d:25-25"));
```

Estos son los distintos tipos de layout dependiendo del tipo de beacon:

ALTBEACON : m:2-3=beac,i:4-19,i:20-21,i:22-23,p:24-24,d:25-25

EDDYSTONE TLM: x,s:0-1=feaa,m:2-2=20,d:3-3,d:4-5,d:6-7,d:8-11,d:12-15

EDDYSTONE UID: s:0-1=feaa,m:2-2=00,p:3-3:-41,i:4-13,i:14-19

EDDYSTONE URL: s:0-1=feaa,m:2-2=10,p:3-3:-41,i:4-20v

IBEACON: m:2-3=0215,i:4-19,i:20-21,i:22-23,p:24-24

Además, podemos diferenciar dentro de un tipo, los beacons que queremos y los que no, gracias al método *RegionBootstrap*. Creamos una *Region* que tiene por parámetros un nombre, y 3 valores que representan *Identifiers.*

```
Region region = new Region("boostrapRegion", null,null,null);
```

El primer String pasado por parámetro, indica un nombre que le vamos a poner a la región, un id personalizado. Después, los siguientes valores son identificadores encargados de diferenciar los beacons. Vamos a poner un ejemplo, si queremos que solo nos detecte un beacon a partir de un UUID, crearíamos la región de esta manera.

```
Region region = new Region("my-beacon-region",
Identifier.parse("0x47814dx6bx5x75x0595b"), null, null);
```

Este sería el caso para un *EddyStone-UID*. Podemos distinguirlos por UUID, por Major y Minor entre otros. Dependiendo del tipo, podemos obtenerlos a partir de una información u otra. En mi caso al usar por defecto *altBeacon*, para crear regiones usamos el UUID para diferenciar los *beacons*. Una vez creada la región, la añadiríamos de esta forma tan simple:

```
regionBootstrap = new RegionBootstrap(this, region);
```

Los últimos 3 métodos del código se encarga de establecer el tiempo entre escaneo, además de una función para reducir el consumo de la batería en segundo plano.

```
@Override
public void didDetermineStateForRegion(int state, Region arg1) {
    estado = state;

    logToDisplay( line: "Current region state is: " + (state == 1 ? "INSIDE" : "OUTSIDE ("+state+")"));

}

@Override
public void didEnterRegion(Region arg0) {
    Log.d(TAG,  msg: "Got a didEnterRegion call");




}

@Override
public void didExitRegion(Region arg0) {

}
```

Figura 23. Tres métodos de la clase BeaconApplicacion.

Los siguientes funciones son los que aparecen en la figura 23, estos se encargan de notificarnos cuando ocurre una acción. En el primer caso, para la función ***DidDetermineStateForRegion*** nos determina en qué estado nos encontramos, ya sea dentro de la región o fuera de la región del *beacon*.

Tiene dos parámetros, el primero, la región que tiene que buscar los *beacons*, y un segundo parámetro, *state,* que tiene valor 0 para cuando el dispositivo se encuentra fuera de la región y valor 1, cuando se encuentra dentro. Hemos diseñado una función para coger este elemento y sea utilizado por otras actividades llamada ***getEstado***, ya que lo vemos bastante útil para el desarrollo de distintos proyectos.

El siguiente caso, es ***didEnterRegion***. Esta función se activa cuando entramos en la región pasada por parámetro, por último, tenemos el caso contrario, ***didExitRegion*** que entra en acción cuando salimos de la región pasada por parámetro, estos dos últimos no se usa en la estructura reutilizable, pero aun así se pueden aprovechar para hacer infinidad de funcionalidades.

En la figura 24 podemos ver el código para mandar una notificación al dispositivo en el momento que queramos.

```
public void notificacionBeacon(Context context, Class<?> cls, String titulo, String subtitulo){
    NotificationCompat.Builder notification;
    notification = new NotificationCompat.Builder(context);
    notification.setAutoCancel(true);
    notification.setSmallIcon(R.drawable.ic_launcher_background);
    notification.setTicker("Bienvenido");
    notification.setPriority(Notification.PRIORITY_HIGH);
    notification.setWhen(System.currentTimeMillis());
    notification.setContentTitle(titulo);
    notification.setContentText(subtitulo);

    Intent intent= new Intent(context,cls);
    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    PendingIntent pendingIntent = PendingIntent.getActivity(context, requestCode: 0,
            intent,PendingIntent.FLAG_UPDATE_CURRENT);
    notification.setContentIntent(pendingIntent);

    NotificationManager nm = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    nm.notify(idNoti,notification.build());
}
```

Figura 24. Método notificationBeacon de la clase BeaconApplication.

El método ***notificacionBeacon*** se encarga principalmente de mandar una notificación al dispositivo, ya sea cuando esté en primer plano o segundo, dirigiéndolo a otra actividad. Tiene cuatro parámetros, el primero es el contexto en el que se encuentra la actividad, el segundo es la clase hacia la que se quiere dirigir, es decir, la actividad que vamos a ver al hacer clic en la notificación. En tercer y cuarto lugar, el titulo y el subtítulo que queremos para nuestra notificación. Por defecto, establecemos una imagen predeterminada, entre otros valores necesarios para la creación de la notificación, para trabajo futuro se puede pasar por parámetro el icono deseado para la creación de la notificación.

La función del método se encarga de inicializar una notificación, crear un *Intent* con la actividad a la que nos queremos redirigir y un gestor de notificaciones encargado de construirlo todo y lanzarla al dispositivo.

Existen otras funciones de menor importancia como ***LogToDisplay*** y ***LogToDisplayOfertas***, encargados de mostrar texto en dos *TextView* predeterminados de la clase *MonitoringActivity*. A estas funciones se le pasan dos valores creados al iniciar la clase llamadas *cumulativeLog y cumulativeLogOfertas* respectivamente. Van acumulando información y cuando el sistema llama a las funciones, se muestran en pantalla. Estas funciones son excelentes para aquellas personas que quieran copiar exactamente la aplicación con sus dos actividades.

Otro método es ***DeleteDisplayOfertas***, encargado de borrar todo lo que escribe la función ***LogToDisplayOfertas***. Y para acabar las funciones ***getLog*** y ***getLogOfertas*** que son las encargadas de devolver los dos valores anteriormente mencionados que acumulan información *cumulativeLog* y *cumulativeLogOfertas.*

4.2.2 MonitoringActivity

Esta actividad es la encargada de monitorizar los distintos *beacons* y responder solo antes los propuestos en nuestra región programada, o a todos los *beacons* si no tenemos creada la región.

```
public class MonitoringActivity extends Activity {

    protected static final String TAG = "MonitoringActivity";

    private static final int PERMISSION_REQUEST_FINE_LOCATION = 1;
    private static final int PERMISSION_REQUEST_BACKGROUND_LOCATION = 2;
    private static final int idNoti = 565656; // NECESARIO PARA notificacionBeacon()



    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_monitoring);
        verifyBluetooth();


        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {...}


    }
```

Figura 25. Parte del código de la actividad *MonitoringActivity*.

En el ***onCreate*** de esta clase, encontramos una función llamada ***verifyBluetooth*** extraída de la aplicación de muestra de la librería de *altbeacon*. Se encarga de verificar que el dispositivo soporta Bluetooth Low Energy (BLE) y que tenemos los permisos para acceder a él para poder activarlo. Justo después tenemos un trozo de código que se encarga de comprobar que versión de API tenemos para comprobar si podemos establecer escaneo en segundo plano, (ya que hasta cierta versión no está permitido) y también, si tenemos permiso para ejecutar la actividad en segundo plano. Es un código bastante extenso pero que no da apenas problemas con muy buena estructuración. Esta parte del código también esta sacado de la librería *altbeacon*.

En mi aplicación prototipo, una vez accedemos a esta actividad, nos aparece, si estamos dentro de la región, las ofertas que tenemos, ese método de solicitar las ofertas se le llama en este método ***onCreate***.

El resto de las funciones son las del ciclo de vida de la actividad, como ***onResume, onDestroy*** y ***onPause*** anteriormente explicadas. Además, dos funciones que se encargan de escribir en dos *TextView* distintos, las funciones encargadas son ***updateLog*** y ***updateLogOfertas*** . Y por último, otra para borrar lo escrito en un *TextView* llamada ***deleteDisplayOfertas***. Estas realmente son utilizadas solo por la clase *BeaconApplicacion* que es la encargada de dirigir todo.

4.2.3 RangingActivity

Esta actividad es la encargada de leer todos los *beacons* uno por uno y coger información de estos, en nuestra aplicación solo cogemos la distancia a la que se

encuentran, dependiendo del tipo podemos coger por ejemplo, el nombre del dispositivo, el nombre del fabricante (por ejemplo Google, Apple, etc). A la hora de crearla solo necesitábamos saber que métodos eran los correctos para implementarla.

```
public class RangingActivity extends Activity implements BeaconConsumer {

    protected static final String TAG = "RangingActivity";
    private BeaconManager beaconManager = BeaconManager.getInstanceForApplication(this);

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_ranging);

    }
```

Figura 26. Método *onCreate* de la clase RangingActivity de la estructura reutilizable.

El método *onCreate*, que vemos en la figura 26, es muy simple, lo único que tenemos que ver, es que se crea una instancia de *beaconManager* al crear la aplicación, donde esta vez se usará para sacar información de los beacons.

```
@Override
protected void onDestroy() { super.onDestroy(); }

@Override
protected void onPause() {
    super.onPause();
    beaconManager.unbind( consumer: this);
}

@Override
protected void onResume() {
    super.onResume();
    beaconManager.bind( consumer: this);
}
```

Figura 27. Funciones de la clase RangingActivity.

Los métodos que vemos en la figura 27, son los necesarios para el ciclo de vida de actividad explicados anteriormente en el punto 4.1. Podemos ver cuando la aplicación deja de estar en primer plano, dejamos de buscar para no gastar batería con el método *beaconmanager.unbind()*, el cual, se encarga de apagar este servicio mientras estamos en segundo plano. Cuando reanudamos vuelve a estar en servicio con el método *beaconManager.bind()* de la función *onResume.*

```
public void onBeaconServiceConnect() {
    RangeNotifier rangeNotifier = new RangeNotifier() {
        @Override
        public void didRangeBeaconsInRegion(Collection<Beacon> beacons, Region region) {
            if (beacons.size() > 0) {
                Log.d(TAG, msg: "didRangeBeaconsInRegion called with beacon count:  "+beacons.size());
                Beacon firstBeacon = beacons.iterator().next();
                logToDisplay( line: "Existe un " + firstBeacon.getParserIdentifier() +
                        " y esta a " + firstBeacon.getDistance() + " metros.");
            }else{
                deleteDisplay();
                logToDisplay( line: "No existen tiendas cercanas en estos momentos");
            }
        }

    };
    try {
        beaconManager.startRangingBeaconsInRegion(new Region( uniqueId: "myRangingUniqueId",
                id1: null,  id2: null,  id3: null));
        beaconManager.addRangeNotifier(rangeNotifier);
        beaconManager.startRangingBeaconsInRegion(new Region( uniqueId: "myRangingUniqueId",
                id1: null,  id2: null,  id3: null));
        beaconManager.addRangeNotifier(rangeNotifier);
    } catch (RemoteException e) {    }

}
```

Figura 28. Función *onBeaconServiceConnect()* de la clase RangingActvity.

La función más importante es ***onBeaconServiceConnect()***, la podemos ver en la figura 28. Se encarga de crear un notificador, de la librería *altbeacon*, la cual tenemos un método *Override* llamado ***didRangeBeaconsInRegion,*** donde sus parámetros son los *beacons* detectados y la región a la que pertenecen. Con este método vamos a ver la información de cada *beacon* gracias a un *Iterator.*

Creamos una sentencia de comprobación, si el tamaño de la colección de *beacons* es mayor que 0, significa que estamos dentro del rango, por tanto, creamos un iterador y vemos por pantalla la información que nos ofrece el *beacon*. Todo esto gracias a otra función creada para ello, prácticamente igual que en la anterior actividad explicada, llamada ***logToDisplay***, en este caso, solo hemos puesto el tipo y la distancia a la que se encuentra. También tenemos la función ***deleteDisplay*** para borrar todo lo escrito con ***logToDisplay***.

Todo esto viene junto a una sentencia try-catch, donde llamamos a *beaconManager,* que es el encargado de inicializar la búsqueda añadiendo la región, podemos añadir distintas regiones, incluso de distintos tipos de *beacon*.

En nuestra aplicación prototipo, esta actividad es la encargada de decirnos a que distancia se encuentra las tiendas si nos encontramos en un centro comercial, incluso con alguna aplicación de terceros podríamos situarlos en un mapa.

4.3 Interfaz JsonApi

Esta interfaz es utilizada para realizar llamadas al servidor, tiene que estar complementada con la librería Retrofit mencionada en las tecnologías utilizadas. En esta

interfaz, lo único que tenemos que realizar es señalar el path y el tipo de método que queremos realizar, ya sea GET, PUT, DELETE o POST.

```
//Devuelve todos los usuarios
@GET("server/usuarios")
Call<List<Usuario>> getNames();
//Devuelve el numero de usuarios
@GET("server/usuarios/count")
Call<Integer> getUsuariosCount();
//Devuelve el usuario pasando el username por parametro
@GET("server/usuarios/username/{user}")
Call<Usuario> getUsuario(@Path("user") String user);
//Devuelve todas las ofertas
```

Figura 29. Métodos de la interfaz JsonApi.

Varios ejemplos los podemos ver en la figura 29. Primero añadimos el método, seguido del path que tiene que seguir la url. Después añadimos la función call y lo que tiene que devolver el servidor. Todo esto va acompañado de este simple método en las clases para realizar las llamadas:

```
Retrofit retrofit = new Retrofit.Builder().baseUrl("https://tfg-ezequiel.appspot.com/")
        .addConverterFactory(GsonConverterFactory.create()).build();
JsonApi jsonApi = retrofit.create(JsonApi.class);
Call<List<Imagen>> call = jsonApi.getImagesByUsername(username);
```

En este método tenemos que crear un *Retrofit.Builder* con la dirección raíz del servidor, añadiendo una Factory para convertir y leer en formato JSON las peticiones y respuestas. Luego, usamos nuestro interfaz con el método que deseemos para conseguir las distintas respuestas del servidor.

4.4 Entidades

En nuestra estructura existen las entidades creadas de nuestra base de datos, que son necesarias para realizar las llamadas al servidor. Pueden ser utilizadas en otros proyectos si están igualmente estructuradas.

```
public class Usuario {

    private int edad;
    private String username;
    private String nombre;

    public int getEdad() {
        return edad;
    }

    public String getUsername() { return username; }

    public String getNombre(){ return nombre;}
}
```

Figura 30. Entidad Usuario de nuestro modelo.

5

Implementación del Servidor

En este capítulo se explica la creación de la API REST, así como los métodos y seguridad, sus servicios, y su posterior subida a la nube de Google Datastore. Este servidor puede ser utilizado de forma complementaria con la estructura común reutilizable si se decide tener una base de datos con las mismas entidades y características.

5.1 Código principal del servidor

5.1.1 Importaciones

El código principal del servidor lo contiene el archivo *main.py* mencionado anteriormente, vamos a explicar lo que debemos importar para realizar un servidor de este tipo con éxito. Lo necesario para que funcione correctamente es:

```
from flask import Flask
app = Flask(__name__)
if __name__ == '__main__':
      app.run('127.0.0.1',80,debug=True)
```

Este código importa *Flask*, y define donde se ejecutará el servidor. Además de importar *Flask*, debemos importar los siguientes paquetes, explicados a continuación:

```
from flask import Flask, render_template
from flask_httpauth import HTTPBasicAuth
from google.cloud import datastore
from datetime import date
from datetime import datetime
import json
import datetime
datastore_client = datastore.Client()
auth = HTTPBasicAuth()
```

- **render_template**: Lo usamos para renderizar la plantilla html.

- **HttpBasicAuth:** Usamos este paquete para la implementación de la seguridad entre la aplicación y el servidor. Es un tipo de seguridad Basic Authentication, donde requiere un usuario y contraseña para acceder a las llamadas. El cliente manda este usuario y contraseña a través de la cabecera.

- **datastore:** Se utiliza para conectarse a la base de datos de Google Datastore.

- **date**: Se utiliza para convertir fechas a un formato que pueda utilizar Google Datastore.

- **json:** Este paquete es uno de los más utilizados ya que se utiliza para convertir las respuestas al cliente en formato JSON.

- **datastore_client:** Es la inicialización de nuestro perfil con el Datastore tendremos que identificarnos en la consola para que sepa el proyecto en que base de datos tiene que buscar.

- **auth:** Inicialización del protocolo Basic Auth.

5.1.2 Métodos root y verify_password

Como vemos en la figura 31, el primer método que tenemos, es devolver simplemente un html, llamado index.html, contiene la vista de nuestro servidor. El método *@app.route('/')* indica la ruta a seguir para ejecutar ese método. En este caso, accede al método root que devuelve una página html renderizada gracias a *render_template*.

En el segundo método tenemos *verify_password(username,password)*, este método es el encargado de comprobar si existe el perfil en la BBDD. Pertenece al paquete *auth* importado anteriormente. Este recibe un usuario y una contraseña codificada a través de la cabecera, se encarga de descodificarlo y comprobar que pertenecen a la base de datos. Si existe devuelve un booleano con valor *True,* en caso contrario, devuelve *False.*

```python
@app.route('/')
def root():
    return render_template('index.html')

@auth.verify_password
def verify_password(username, password):
    query = datastore_client.query(kind='Usuarios')
    query.add_filter('username','=',username)
    query.add_filter('password','=',password)
    lista = list(query.fetch())
    if len(lista) > 0:
        return True
    else:
        return False
```

Figura 31. Dos métodos de nuestro código del archivo main.py.

Vemos que para comprobar el usuario y la contraseña, lo único que tenemos que hacer es realizar una búsqueda en la base de datos (*query*), añadiendo los filtros necesarios para diferenciar los parámetros.

5.1.3 Métodos GET de Usuarios

Vamos a explicar cómo hemos desarrollado los métodos para la entidad Usuarios de nuestra base de datos. Estos métodos son aplicables al resto de entidades.

En primer lugar vamos a explicar el más sencillo de todos, que es la devolución de todos los usuarios de la BBDD. En la figura 32 podemos ver el código de este método.

```python
@app.route('/server/usuarios/', methods=['GET'])
@auth.login_required
def usuarios():
    query = datastore_client.query(kind='Usuarios')
    return json.dumps(list(query.fetch()))
```

Figura 32. Método GET que devuelve todos los usuarios de la BBDD.

La ruta para este método sería https://tfg-ezequiel.appspot.com/server/usuarios/ por tanto la *app.route* será la indicada en la figura 26, '/server/usuarios/'. Además, también tenemos que indicar que tipo de llamada responde este método, en este caso será solo GET.

Después tenemos *@auth.login_required*, esta línea da seguridad al método, es decir, para acceder a este método, en la cabecera deberá contener la autenticación (usuario y contraseña), si pertenecen a la BBDD, se otorga permiso y podrá leer lo que contiene el método.

Justo después empieza la definición del método, donde primero, realizamos una búsqueda en la BBDD (*query)* buscando todos los usuarios. Estos usuarios se cogen con el método *query.fetch*(), los guardamos en una lista, luego se convierte en formato JSON gracias al método *json.dumps*() y se devuelven al cliente.

Es el método más sencillo que nos podemos encontrar en esta API REST. A continuación veremos un método donde pasamos un usuario por parámetro y nos devuelve ese usuario.

```
@app.route('/server/usuarios/username/<usuario>/', methods=['GET'])
@auth.login_required
def usuariosByUsername(usuario):
    query = datastore_client.query(kind = 'Usuarios')
    query.add_filter('username', '=', usuario)
    lista = list(query.fetch())
    if len(lista) > 0:
        return json.dumps(lista[0])
    else:
        return json.dumps(lista)
```

Figura 33. Método que devuelve un usuario a partir de un username.

Este método es muy parecido al anterior, la única diferencia es que se le pasa por parámetro un *username.* En *@app.route* ponemos entre <> el parámetro que queremos enviar. En la definición del método, el parámetro debe llamarse igual, para así, poder usarlo a la hora de realizar la *query.* Realizamos la petición a la BBDD, pero esta vez, añadiendo un filtro, el cual, el valor *username* de la BBDD de la entidad *usuarios*, tiene que ser igual al parámetro que le hemos pasado. El resultado de la petición lo guardamos en una lista llamada *lista* con *list(query.fetch())*. Si esta *lista* tiene tamaño mayor de 0, devolvemos el primer elemento de la lista (lógicamente solo debería de haber un elemento en la lista ya que no pueden existir más de un usuario con ese *username*).

5.1.4 Métodos PUT y DELETE

```python
@app.route('/server/usuarios/<id>/', methods=['GET', 'PUT', 'DELETE'])
@auth.login_required
def usuario(id):
    if request.method == 'DELETE':
        key = getKey('Usuarios', id)
        if key != None:
            datastore_client.delete(key)
        return ('', 200)
    elif request.method == 'PUT':
        data = request.json
        with datastore_client.transaction():
            key = getKey('Usuarios', id)
            if key != None:
                key = datastore_client.key('Usuarios', key.id)
                user = datastore_client.get(key)

                user['username'] = data['username']
                user['nombre'] = data['nombre']
                user['edad'] = data['edad']
                user['email'] =data['email']

                datastore_client.put(user)
        return ('', 200)
```

Figura 34. Método que borra o actualiza un usuario a partir de un ID.

Vemos en la figura 34 como se realizan llamadas PUT y DELETE a la hora de interactuar con la BBDD. El método DELETE, es muy simple. Conseguimos la key relacionada con el id del usuario que queremos borrar, ya que para borrar o modificar solo se puede buscar por key en la BBDD. Una vez tengamos esa key, lo único que tenemos que hacer es llamar al método *datastore_client.delete* con el parámetro de la key para borrarlo de la BBDD.

En cuanto al método PUT, primero tenemos que leer los parámetros que nos ha mandado el cliente y éste deberá mandar estos datos en JSON. Nosotros lo recuperaremos gracias a *request.json* y *datastore.client_transaction()*. Cogemos la key del usuario que queremos cambiar y llamamos a *datastore_client.get(key)* para recuperar ese usuario, después cambiamos los parámetros por los recibimos en formatos JSON por el cliente, y terminamos actualizando la BBDD llamando al método *datastore_client.put.*

Todos estos métodos explicados en este punto y en el punto anterior son aplicables al resto de entidades.

5.2 Subida a la nube Google Datastore

Para empezar a usar este servicio, lo primero que tenemos que hacer es habilitar la facturación, el primer año es gratis si no se supera la cuota indicada. Después necesitamos crear un proyecto en la consola de Google, a través de la URL, https://console.cloud.google.com/, a continuación descargamos el SDK de Google Cloud.[16]

Una vez creada la cuenta, necesitamos habilitar la API de Google Cloud dentro de nuestra cuenta, para poder hacer test, construir y desplegar proyectos.

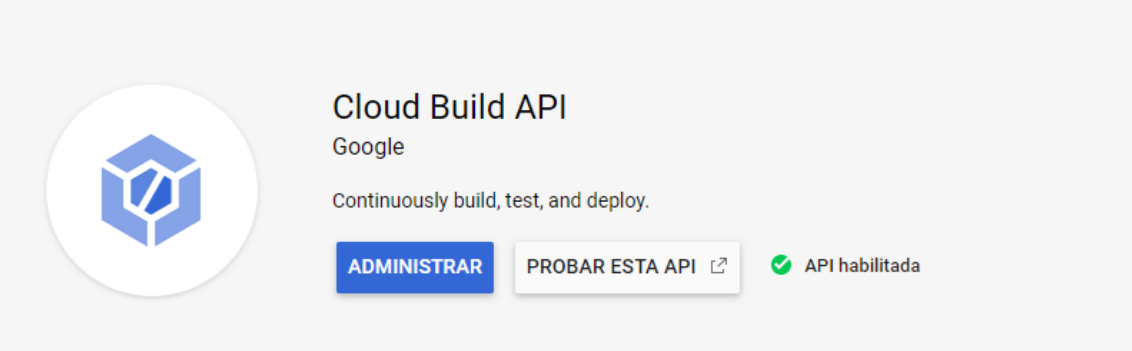

Figura 35. API para desplegar nuestro proyecto. Imagen sacada de la web, https://console.developers.google.com/apis/library/cloudbuild.googleapis.com

Como estamos usando *Python*, necesitamos una estructura necesaria que es la siguiente:

- NombreServidor/
  - App.yaml
  - Main.py
  - Requirements.txt
  - Static/
    - Scripts.js
    - Style.css
  - Templates/
    - Index.html

- **NombreServidor**: Esta sería la carpeta inicial y puede tener un nombre cualquiera, contendrá todos los archivos necesarios para la creación del servidor. Es el directorio raíz del proyecto.

- **App.yaml:** Este archivo especifica cómo las rutas de URL se corresponden con los controladores de solicitud y los archivos estáticos además de contener información sobre el código de la app como el entorno de ejecución. Tiene un formato YAML.[17]

```yaml
runtime: python37

handlers:

- url: /static
  static_dir: static


- url: /.*
  script: auto
  secure: always

```

Figura 36. Estructura del archivo app.yaml con formato YAML de nuestro servidor creado.

- **Main.py:** En este archivo es donde se escribe el código, lo enseñaremos más adelante.

- **Requeriments.txt:** Es el archivo que contiene las dependencias que tiene el proyecto, se puede saber las dependencias escribiendo en la consola → pip freeze.

```
Flask==1.1.1
flask-httpauth==4.0.0
requests==2.22.0
google-api-python-client==1.7.11
google-cloud-datastore==1.10.0
```

Figura 37. Estructura del archivo requeriments.txt de nuestro servidor.

- **Static:** Esta carpeta contiene archivos JavaScript o archivos CSS de nuestros archivos html, es decir los comportamientos y estilos de los archivos.

- **Templates:** Por último esta carpeta contiene los archivos html de nuestro servidor. Nuestro archivo html es algo simple donde nos señala los distintos tipos de llamadas que se pueden hacer al servidor, es la vista que aparece al escribir la dirección de nuestro servidor: https://tfg-ezequiel.appspot.com/. En la figura 38 y 39 se puede ver el archivo html de nuestro servidor y el resultado en un navegador web.

```html
<html>

<head>
    <title>API REST - TFG Ezequiel</title>
    <meta charset="UTF-8">
    <link rel="shortcut icon" type="image/png" href="{{ url_for('static', filename='images/bikesharing.png') }}" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="google-signin-client_id" content="555443826380-o8c2e2as9b3qp327cafgi0mpajogeq63.apps.googleusercontent.com">
    <script src="https://apis.google.com/js/platform.js" async defer></script>
</head>

<body>
    <h1>API REST TFG - Ezequiel Rodriguez</h1>
    Distintas llamadas al servidor:<br>
    <a href="https://tfg-ezequiel.appspot.com/server/usuarios/">Usuarios<br></a> GET server/usuarios/ <br>
    GET server/usuarios/count <br>
     GET server/usuarios/username/{username} <br><br>
    <a href="https://tfg-ezequiel.appspot.com/server/ofertas/">Ofertas<br></a> GET server/ofertas/ <br>
     GET server/ofertas/username/{username} <br><br>
    <a href="https://tfg-ezequiel.appspot.com/server/citas/">Citas<br></a> GET server/citas/ <br>
    GET server/citas/username/{username}<br>
     GET server/citas/date/{day}/{month}/{anyo}<br> GET server/citas/date/today/ <br> <br>
    <a href="https://tfg-ezequiel.appspot.com/server/imagenes/">Imagenes<br></a> GET server/imagenes/ <br>
     GET server/imagenes/username/{username}<br><br>
    <a href="https://tfg-ezequiel.appspot.com/server/mapas/">Mapas<br></a> GET server/mapas/<br>
    GET server/mapas/username/{username} <br><br>
    <a href="https://tfg-ezequiel.appspot.com/server/productos/">Productos</a><br> GET server/productos/ <br>
     GET server/productos/username/{username} <br>
</body>
</html>
```

Figura 38: Estructura de nuestro único archivo index.html de la carpeta templates.

API REST TFG - Ezequiel Rodriguez

Distintas llamadas al servidor:
Usuarios
GET server/usuarios/
GET server/usuarios/count
GET server/usuarios/username/{username}

Ofertas
GET server/ofertas/
GET server/ofertas/username/{username}

Citas
GET server/citas/
GET server/citas/username/{username}
GET server/citas/date/{day}/{month}/{anyo}
GET server/citas/date/today/

Imagenes
GET server/imagenes/
GET server/imagenes/username/{username}

Mapas
GET server/mapas/
GET server/mapas/username/{username}

Productos
GET server/productos/
GET server/productos/username/{username}

Figura 39. Html del servidor web. - https://tfg-ezequiel.appspot.com/

Por último, con todo creado ya, lo único que tendríamos que hacer, es usar la herramienta *gcloud* del SDK de Google Cloud para implementar el servicio web en la nube App Engine. Tenemos que usar el comando "*gcloud app deploy*" desde el directorio

raíz de tu proyecto, en el que se encuentra el archivo app.yaml. La aplicación se desplegará en la dirección que pone en *target url* mostrada en la figura 40.

```
PS C:\Users\Ezequiel\Documents\VisualStudioCodeProjects\servidorTFG-Ezequiel> gcloud app deploy
Services to deploy:

descriptor:      [C:\Users\Ezequiel\Documents\VisualStudioCodeProjects\servidorTFG-Ezequiel\app.yaml]
source:          [C:\Users\Ezequiel\Documents\VisualStudioCodeProjects\servidorTFG-Ezequiel]
target project:  [tfg-ezequiel]
target service:  [default]
target version:  [20200603t134921]
target url:      [https://tfg-ezequiel.appspot.com]


Do you want to continue (Y/n)?  y

Beginning deployment of service [default]...
#============================================================#
#= Uploading 1 file to Google Cloud Storage                 =#
#============================================================#
File upload done.
Updating service [default]...done.
Setting traffic split for service [default]...done.
Deployed service [default] to [https://tfg-ezequiel.appspot.com]
```

Figura 40. Código que aparece al ejecutar el comando *gcloud app deploy.*

Gracias a VisualCodeStudios podemos ejecutar nuestro servidor ejecutando en la consola de VCS: python main.py runserver 80. El último número indica en que puerto se despliega el servidor, en nuestro caso es 127.0.0.1/80, y veríamos nuestro servidor en navegador.

6

Conclusiones y trabajo futuro

Se ha realizado el proyecto en tres distintas iteraciones donde se ha ido mejorando y añadiendo más funcionalidades poco a poco. En cuanto a la aplicación prototipo se ha realizado especificación de requisitos, estudio las de tecnologías utilizadas, implementación de la aplicación en Android usando nuestro propio esqueleto, implementación del servidor en Python y el diseño y desarrollo de una base de datos. Esta estructura permite tener un servidor de BBDD diferente, además de la posibilidad de mejora de una aplicación cliente a partir de la propuesta.

Han existido dificultades al principio, sobre todo con la aplicación prototipo y la estructura, por ejemplo, no encontrar la manera de integrar la búsqueda de beacons con las llamadas al servidor. En cuanto al resto, en el mapeo de la base de datos con los valores enviados al servidor, ya que Google Datastore no permitía ciertos valores, entre otros problemas.

Cuando empecé este proyecto, no tenía ningún conocimiento de Android, y a día de hoy puedo realizar una aplicación estándar en este lenguaje sin ningún problema. Seguir los pasos de ingeniería del software que he aprendido durante estos años, como diagrama casos de uso, diagrama de secuencia entre otros, además del uso del patrón de Modelo-Vista-Controlador, me han ayudado mucho a avanzar de manera constante en este proyecto, donde la dificultad crecía en algunos momentos donde no sabía por dónde empezar a investigar, ya que no conocía el uso de beacons y creo que tiene un gran potencial en cualquier sector, tiene infinidad de posibilidades y en un futuro se verán más proyectos con estos dispositivos.

Este proyecto me ha hecho emplear distintos conocimientos de distintas asignaturas, incluso desarrollar un servidor en un lenguaje que no dominaba, realizar una aplicación cliente en Android entre otras cosas. Además muestra la importancia de

la tecnología de proximidad a través de la interacción con el ámbito que nos rodea, aprovechando recursos como puede ser Bluetooth y su versión de bajo consumo energético (*low energy*), permite una comunicación más eficiente, esto queda claro cuando grandes compañías desarrollan sus propios protocolos como Apple o Google.

6.1 Trabajo futuro

En el caso que se siguiera desarrollando este proyecto existen distintas mejoras que se pueden implementar:

- **Integrar terceras aplicaciones.** El hecho de integrar mapas o uso de imágenes de alguna API de terceros ayudaría mucho a establecer posicionamientos de los clientes frente a los beacons entre otras funcionalidades.

- **Seguridad cliente-servidor.** La seguridad de mi aplicación prototipo es baja comparado con las posibilidades que hay, ya que es la más básica y no permite terceras aplicaciones. El uso de OAuth 2.0 sería importante de aplicar, además que Google ofrece distintos códigos de ayuda para ello.

- **Optimización de la estructura y aplicación de prueba.** Mejorar la estructura del esqueleto además de añadir distintas funciones. En la aplicación de prueba mejorar los distintos aspectos y convendría hacer un tratamiento más profundo de los posibles errores que pueden producirse y desarrollar mecanismos de prevención para ellos. Integrar inicio de sesión con Facebook, posibilidad de registro de usuario o de cambio de contraseña.

- **iOS**: Desarrollar la estructura en iOS.

- **Posibilitar el ajuste de parámetros.** A la hora de buscar los beacons se puede, por ejemplo, establecer distintos tiempos de escaneo.

Referencias

[1] What is a Beacon? Accent Systems, 2018. Accent Systems [online], https://accent-systems.com/es/support/faq/what-is-a-beacon/

[2] Specification for Eddystone, an open beacon format from Google GITHUB, google/eddystone, 2020, GitHub [online], https://github.com/google/eddystone

[3] Eddystone format | Beacons | Google Developers, 2018. Google [online], https://developers.google.com/beacons/eddystone

[4] Getting Started with iBeacon, 2020, Apple, [online]. https://developer.apple.com/ibeacon/Getting-Started-with-iBeacon.pdf

[5] Conoce Android Studio | Android Studio. Developer.android.com[online]. 2020.

https://developer.android.com/studio/intro/index.html?hl=es-419

[6] Getting Started with Gradle, 2020. Petri Kainulainen[online], https://www.petrikainulainen.net/getting-started-with-gradle/

[7] Proyectos recientes | Android Studio, 2020. Developer.android.com [online], https://developer.android.com/studio/projects/index.html?hl=es-419

[8] Libreria RetroFit para Android y Java |[online] https://square.github.io/retrofit/

[9] Quickstart Google Cloud Datastore [online] https://cloud.google.com/datastore/docs/quickstart#create_a_database

[10] ¿Qué es Python?[online],2020. https://web.archive.org/web/20200224120525/https://luca-d3.com/es/data-speaks/diccionario-tecnologico/python-lenguaje

[11] Why was Python created in the first place?. General Python FAQ. [online]

https://docs.python.org/3/faq/general.html#why-was-python-created-in-the-first-place

[12] PEP 20-The Zen of Python. https://www.python.org/dev/peps/pep-0020/

[13] Que es Flask. Blog OpenWebinars, 2017. [online] https://openwebinars.net/blog/que-es-flask/

[14] MVC (Modelo-Vista-Controlador): ¿qué es y para qué sirve?. 2017. [online]. https://codingornot.com/mvc-modelo-vista-controlador-que-es-y-para-que-sirve

[15] Notes and Historical Documents.[online]

http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[16] Compila una app de Python 3.7 en App Engine. Google Cloud. [online]

https://cloud.google.com/appengine/docs/standard/python3/building-app/

[17] Referencia de app.yaml. Documentación Google Cloud. [online]

https://cloud.google.com/appengine/docs/standard/python/config/appref?hl=es-419

[18] Google. Android Developers - Activities. [online] https://developer.android.com/guide/components/activities/intro-activities

[19] Android Beacon Library. [online] https://altbeacon.github.io/android-beacon-library/index.html

[20] ¿Qué es beacon manager y para qué sirve?. [online]

http://blueservicesa.com/beacons/que-es-beacon-manager-y-para-que-sirve/#:~:text=Beacon%20manager%20es%20una%20aplicaci%C3%B3n,Esta%20aplicaci%C3%B3n%20te%20permite%3A&text=Tocar%20para%20agregar%3A%20Permite%20agregar%20r%C3%A1pidamente%20beacons%20cercanos%20a%20tu%20cuenta.

Apéndice A

Manual de Instalación

Este apartado será exclusivo para la información de la importación de la librería y su correcto funcionamiento. Para empezar necesitamos crear un proyecto en Android Studio:

1. File -> New -> New Project -> Next -> Elegimos un nombre para nuestro proyecto, elegimos en *Language Java*, y la API mínima del proyecto que será API 21.-> Finish.

2. File-> New ->New Module->Seleccionamos Import .JAR/.ARR Package -> Next -> buscamos el directorio donde se encuentra el archivo de la librería .ARR (situada en la carpeta Output dentro de la carpeta build de la librería) -> Finish.

3. Escribir la siguiente línea en el Gradle del módulo de la aplicación:

```
implementation project(path: ':*nombre_archivoAAR*')
```

por ejemplo para un archivo con nombre libbeacon-release.aar:

```
implementation project(path: ':libbeacon-release')
```

4. Ya tenemos la librería correctamente importada, para que funcione completamente habría que implementar una par de líneas en el Gradle del módulo aplicación:

```
implementation                'androidx.legacy:legacy-support-v4:1.0.0'
implementation                'org.altbeacon:android-beacon-library:2+'
implementation                 'com.squareup.retrofit2:retrofit:2.8.1'
implementation 'com.squareup.retrofit2:converter-gson:2.8.1'
```

Estas líneas sirven para usar las librerías Retrofit con su factory Gson además de la librería *altbeacon*.

5. Extender una clase a BeaconApplication e implementar los Override Method, y añadir en el manifiesto de Android: Android-name: .*nombre clase que se ha extendido*.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga