Universidad de Extremadura

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Software

Trabajo Fin de Grado

SmartFidelity

Adrián Mateos Pérez

Junio, 2016

Universidad de Extremadura

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Software

Trabajo Fin de Grado

SmartFidelity

Autor: Adrián Mateos Pérez

Fdo.:

Director: José María Conejero Manzano

Fdo.:

Director: Roberto Rodríguez Echeverría

Fdo.:

Tribunal Calificador

Presidente:

Fdo:

Secretario:

Fdo:

Vocal:

Fdo:

RESUMEN

Nuestra sociedad se está habituando cada vez más al uso de aplicaciones móviles. En los últimos años hemos notado un gran aumento de las personas que utilizan este tipo de dispositivos. Este incremento de la aceptación de estos dispositivos nos permite abrir nuevos mercados, y va a producir un cambio en la forma en que las empresas fidelizan a sus clientes. En este ámbito es en el que se centrará este proyecto: cambiar la forma en la que las empresas fidelizan a sus clientes gracias a la información contextual que proporciona un dispositivo móvil y su ubicación.

En este documento se explicarán los pasos que se han seguido para **desarrollar una aplicación móvil que tiene como objetivo mejorar la fidelización de clientes** y que, gracias al uso de **Beacons**, nos permitirá conocer mejor a nuestros clientes y ofrecerles lo que quieren en cada momento.

Como en todo proyecto, lo primero que se ha hecho es un estudio de qué **tecnologías son las más apropiadas** para conseguir que la solución final sea la más adecuada, pensando que en un futuro este proyecto se pueda ampliar y mejorar. También se detalla en profundidad estas tecnologías, en qué consiste cada una de ellas y qué es lo que aportan al proyecto.

CONTENIDO

ILUSTRACIONES

1. INTRODUCCIÓN

1.1. DESCRIPCIÓN DEL PROYECTO

En este proyecto se ha abordado la creación de una aplicación móvil que permita a los establecimientos **mejorar la fidelización** de sus clientes, gracias a la información de consumo de estos, así como **unificar los sistemas tradicionales de fidelización** (Tarjetas, cupones descuento, ...).

La idea es que el usuario haga uso de la misma aplicación para todos los establecimientos, que vaya acumulando puntos por sus compras y obteniendo recompensas por esos puntos. **El establecimiento podrá llamar la atención de los usuarios** que utilicen la aplicación **gracias al sistema de notificaciones push y al uso de dispositivos Beacons,** que les permitirá llamar su atención en el momento justo.

Además, para **facilitar e incentivar el consumo de los clientes en los establecimientos**, se ha puesto especial atención a mejorar el **registro de las compras**, permitiendo que el usuario registre sus compras de forma inalámbrica mediante el uso de la aplicación, haciendo uso para ello de **tecnología NFC** o mediante la lectura de **códigos QR**.

1.2. MOTIVACIÓN DEL PROYECTO

La idea de este proyecto surge de la **necesidad real de un establecimiento de Cáceres** que veía en los dispositivos móviles una posibilidad real de mejorar su sistema de fidelización, añadiendo "inteligencia" a este sistema para conocer mejor a los clientes y automatizar ciertos procesos. Además de esta necesidad real que ya conocíamos, pensamos que se pueden mejorar mucho **los sistemas tradicionales de fidelización** gracias al potencial que ofrecen los dispositivos móviles y diversas tecnologías seleccionadas para cubrir este proyecto**.**

Actualmente, cada establecimiento tiene su propio sistema de fidelización, con el alto coste que esto representa (integración en el sistema, impresión de tarjetas de fidelización, impresión de anuncios...). Este proyecto busca **unificar todos los sistemas de fidelización en uno solo**, que permita a los establecimientos reducir sus gastos en este campo y mejorar la atención que prestan sus clientes a sus campañas.

Con esta premisa surge la idea de crear una aplicación móvil que, gracias a la gran acogida que tienen los Smartphone en nuestra sociedad, nos permitirá unificar todos los sistemas en uno solo. El uso de **Beacons** surgió por su **gran potencial en este ámbito**, ya que nos permite tener **información contextual de nuestros usuarios** y nos permitirá seleccionar la oferta a mostrarle que más le pueda interesar según su ubicación.

1.3. OBJETIVOS

El objetivo de este proyecto es claro, desarrollar una aplicación que permita unificar y mejorar la fidelización de clientes, pero para hacerlo hay que tener en cuenta muchos aspectos y tomar muchas decisiones de cómo hacerlo. Para ello, se han definido la siguiente lista de sub-objetivos:

- Conseguir que la solución sea lo más económica posible
- Que los datos sean accesibles desde cualquier dispositivo (almacenamiento en la nube)
- Que el usuario pueda utilizar la aplicación sin conexión puntualmente (modo offline)
- Que el establecimiento pueda establecer anuncios y asociarlos a un Beacon, y que lleguen al usuario al estar cerca de ese Beacon.
- Que el cliente pueda registrar una compra de forma sencilla.
- Que el cliente pueda consultar las promociones disponibles.

2. BUSINESS CANVAS MODEL

En esta sección se ha desarrollado un mini plan de negocio para demostrar la viabilidad real del proyecto. El objetivo era analizar si la idea desarrollada en el proyecto podría ser implantada en un entorno real. Para ello, se ha desarrollado un **Business Canvas Model** en el que se muestran las claves para entender esta posible idea de negocio [69].

The Business Model Canvas

Designed for: SmartFidelity | *Designed by:* Adrián Mateos Pérez | *Date:* 03/04/2016 | *Version:* 1.0

Key Partners

- Inversores que apuesten por nuestro sistema
- Firebase
- Proveedor de flyers y pegatinas identificativas para los establecimientos
- Proveedor de Beacons
- Redes Sociales e Influencers para dar a conocer el producto
- Se buscarían asociaciones con grandes marcas que puedan estar interesadas en nuestro producto para promocionarlo ofreciendoles alguna ventaja, por ejemplo, El corte inglés, Carrefour, Inditex...

Key Activities

- Mejora continua del producto para ofrecer más valor a los usuarios y adaptarnos al futuro de este sector
- Estudio de los hábitos de los clientes para ofrecer mejores recomendaciones a los establecimientos y mejorar la fidelización de los clientes
- Soporte a los usuarios del producto
- Estudio continuo de los anuncios y promociones de los clientes para mejorar las ventas y así conseguir mayores ingresos

Key Resources

- Financiación de inversores
- Firebase - Para el almacenamiento, autenticación y gestión de usuarios
- Personal - Un desarrollador para mejorar la aplicación y realizar las tareas de marketing online y venta del sistema
- Portátil - Para poder desarrollar las tareas de la empresa
- Publicidad - Tanto online como flyers
- Beacons - Para poder enviárselos a los nuevos establecimientos
- Pegatinas identificativas para enviárselos a los nuevos

Value Propositions

- SmartFidelity ofrece a los propietarios una forma de mejorar la fidelización de sus clientes gracias a la utilización de una aplicación móvil que les permite establecer anuncios y promociones de sus productos que les llegarán al usuario en el mejor momento (cuando están cerca del establecimiento) promoviendo el consumo y las ganancias del establecimiento
- Con SmartFidelity el establecimiento se puede olvidar de los sistemas tradicionales de fidelización basados en una tarjeta de usuario o similar y apostar por un sistema global en el que accederá a todos los usuarios de la plataforma y los podrá convertir en sus clientes atrayéndolos mediante promociones
- Para los clientes de los establecimientos, SmartFidelity les aportará el poder olvidarse de las tarjetas de fidelización y llevar todas en un único lugar, la aplicación. Además, le permitirá acceder a un montón de promociones exclusivas.
- Otro valor de SmartFidelity es que no existe ningún sistema igual actualmente, ya que no existe ningún sistema de fidelización que permita englobar a todos los usuarios de distintos establecimientos y a todos los establecimientos en un único sistema de fidelización.

Customer Relationships

- La relación con el cliente será principalmente mediante la aplicación y una web donde podrán conocernos y contratar el servicio. Tambien pondremos a su disposición medios para contactarnos y poder resolver sus posibles dudas o problemas
- Mediante la aplicación un propietario podrá darse de alta en el sistema y tras la realización del pago inicial se le enviarán los Beacons necesarios y la pegatina identificativa de que dispone del sistema SmartFidelity en su establecimiento
- Los usuarios podrán registrarse mediante la aplicación e igualmente contactar con nosotros para posibles dudas o problemas, para ellos el sistema será gratuito

Channels

- Para darnos a conocer se utilizará tanto el marketing online como la búsqueda de nuevos establecimientos mediante reuniones con los propietarios
- Para los usuarios, se atraerán mediante marketing online, los carteles del establecimiento que indican que dispone de éste sistema y mediante boca a boca de otros usuarios (WOMM)

Customer Segments

- Propietarios de cualquier tipo de establecimiento (Tiendas de ropa, grandes almacenes, Bares, Restaurantes...) con necesidad de mejorar la fidelización de su negocio y/o crear un sistema de fidelización aprovechando las ventajas de SmartFiedlity
- Clientes de las tiendas en las que esté disponible el sistema y que estén interesados en tener buenas ofertas y olvidarse de los sistemas de fidelización tradicionales. Para estos usuarios el sistema será totalmente gratuito promoviendo su uso en masa.

Cost Structure

- Firebase
  - Se podría empezar con la versión gratuita hasta superar sus límites (100 conexiones simultáneas, 1GB de almacenamiento y 10GB de transferencia). Cuando se supere ese límite se pasaría a pagar 50€ mensuales con conexiones ilimitadas, 10GB de almacenamiento y 50GB de transferencia.
- Personal - Couta de autonomo (aproximadamente 150€ mensuales)
- Gastos de equipamiento - Gasto inicial para Portátil (aproximadamente 500€)
- Publicidad en redes sociales y flayers (aproximadamente 50€ mensuales)

Revenue Streams

- Cuota a pagar por los propietarios de los establecimientos cada trimestre formada por:
  - Cuota fija por utilización y soporte (20-40€)
  - Cuota variable según rendimiento de los anuncios y ventas (0.02€ por anuncio visualizado y 0.20€ por venta promovida por un anuncio)
- Coste inicial a pagar por el propietario del establecimiento para poder comenzar a utilizar el sistema y que constará de la creación de la cuenta (en principio sin coste) y la instalación de los Beacons necesarios (20€ por Beacon)

DESIGNED BY: Strategyzer AG
The makers of Business Model Generation and Strategyzer

Strategyzer
strategyzer.com

2.1. POSIBLES COMPETIDORES

Se han encontrado proyectos cuyo objetivo es similar al de **SmartFidelity**. Estos proyectos se presentan a continuación:

- **Nubbler** → Esta aplicación parece que va a ser muy similar a **SmartFidelity**, ya que su objetivo es utilizar los **Beacons** para enviar a los clientes promociones en el momento justo, además de llevar la **fidelización de los clientes [68]**. La noticia de esta nueva aplicación ha salido en marzo de 2016 y todavía está en pruebas, pero podría ser uno de los principales competidores por ser la misma idea. De este proyecto solo se conoce la información publicada en la noticia de *laopiniondemalaga.es*, por lo que no se conoce mucho de sus opciones y, por tanto, no se puede comparar todavía con **SmartFidelity**.

- **Mi Carrefour** [3] → Esta aplicación creada por **Carrefour** para sus clientes es quizás lo más parecido a la funcionalidad que se quiere conseguir con **SmartFidelity** (tarjetas de fidelización electrónicas, tiendas, folletos promocionales, compras…), pero con la ventaja de que **SmartFidelity** permite localización de los clientes e interactuar con éstos según su localización gracias a los **Beacons** (funcionalidad no disponible en la aplicación de **Carrefour**). Otra ventaja de **SmartFidelity** sería que no solo sería una aplicación para **Carrefour**, sino que los usuarios podrían utilizarla para todos sus establecimientos habituales. También **SmartFidelity** podría ofrecer en un futuro la integración de pagos con **Android Pay**.

- **Fidelizoo** [4] → Esta aplicación se basa en que los clientes se registran en su plataforma y pueden ganar puntos por compras en establecimientos asociados a **Fidelizoo**. Para registrar las compras **Fidelizo** permite utilizar un **código QR** que genera la aplicación, utilizar la **tarjeta de Fidelizoo impresa con el código QR** o **introducir el mail o número de teléfono** registrado por el cliente. Esta es la misma idea que se busca conseguir en **SmartFidelity** pero añadiendo el registro con **NFC** y las ventajas de utilizar **Beacons**.

[3] Nueva APP mi Carrefour - Descubre todas las novedades https://www.youtube.com/watch?v=SMBedJO4Yss [Disponible a 10 de abril de 2016]

[4] Fidelizoo http://www.fidelizoo.com/

- **Makeitapp**[5] → Esta es una empresa dedicada a crear aplicaciones para establecimientos que quieran tener su aplicación de fidelización de clientes utilizando las ventajas que ofrecen los Beacons para fidelizar clientes. Esta aplicación podría tener algunas similitudes con **SmartFidelity**, pero al ser específica para cada establecimiento no tendría el mismo potencial y sería más costosa para los establecimientos.

- **Stocard, NoKadi, FidMe, Fidiliti** → Estas aplicaciones se encargan solamente de la función digitalización de las tarjetas de fidelización, almacenando el código asociado a cada una y ahorrándonos espacio en la cartera [67]. Esta función la realizaría **SmartFidelity** ya que el usuario solo se identificaría de forma global en todas las tiendas asociadas con solo identificarse en la aplicación de **SmartFidelity.**

- **Tiendeo** → Esta aplicación se centra única y exclusivamente a ofrecer los folletos de las tiendas de todo tipo de establecimientos dentro de una aplicación. De esta forma, en la aplicación se podrían visualizar las últimas ofertas del McDonald, Burger King, Zara, Carrefour… Este concepto también se ha tenido en cuenta en **SmartFidelity** ya que la aplicación permitiría a los establecimientos publicar sus ofertas y los clientes podrían consultarlas.

[5] Markeitapp http://www.makeitapp.com/es/ibeacon

3. TECNOLOGÍAS UTILIZADAS

3.1. ANDROID

3.1.1. ¿QUÉ ES ANDROID?

Android es un **sistema operativo para dispositivos móviles** basado en Linux que es propiedad de Google [1]. Este sistema operativo está pensado para trabajar con dispositivos **móviles con pantalla táctil**, con un **gran número de sensores** y **conexión a internet**.

Este sistema operativo fue inicialmente desarrollado por Android Inc, una pequeña empresa que tenía como objetivo desarrollar un sistema operativo basado en Linux. Google apoyó económicamente a Android Inc desde sus inicios, y fue en 2005, cuando Android Inc terminó siendo adquirida por Google, que vio el potencial que podría tener en un futuro este sistema operativo, y ha apostado muy fuerte por él hasta convertirlo en el principal sistema operativo del mundo para dispositivos móviles.

Android fue presentado en 2007 junto a la fundación del **Open Handset Alliance** (un consorcio de compañías de hardware, software y telecomunicaciones) para avanzar en los estándares abiertos de los dispositivos móviles.

Android tiene un proyecto Open Source llamado **Android Open Source Project (AOSP)**, cuyo objetivo es compartir el código y permitir a los desarrolladores sacar sus propias versiones modificadas de este sistema operativo, gracias a este proyecto salen las principales ROMs de Android, por ejemplo, Cyanogen.

3.1.2. ARQUITECTURA ANDROID

La siguiente imagen muestra la pila arquitectónica de Android donde podemos localizar los principales componentes que definen este sistema operativo.

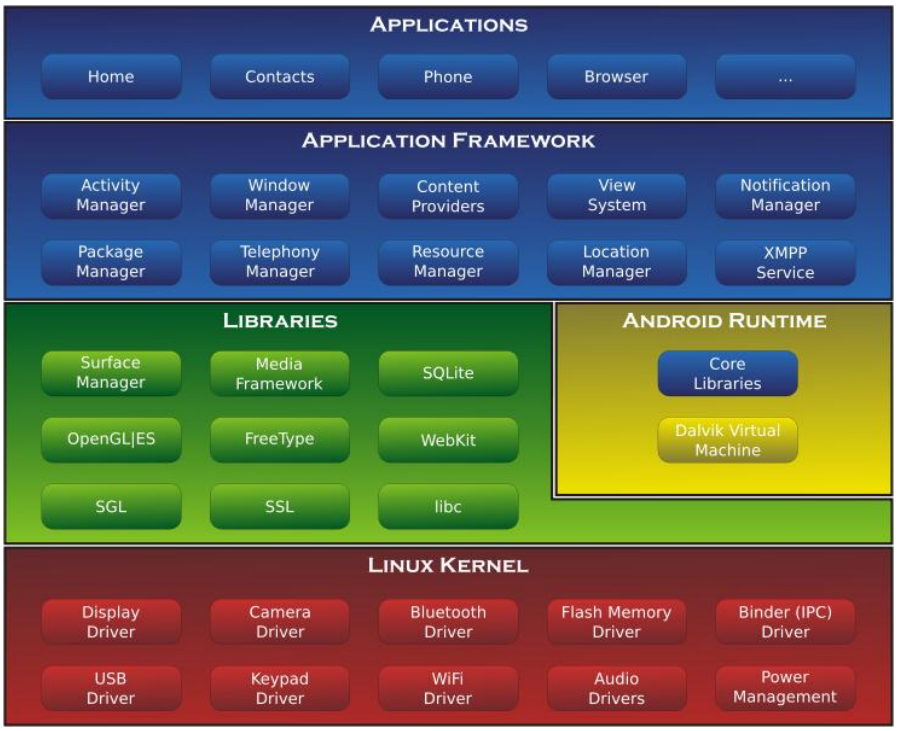


Ilustración 1 - Arquitectura Android

En el primer nivel tenemos **el núcleo Linux**, donde se encuentran la base de servicios del sistema como seguridad, gestión de memoria, gestión de procesos, pila de red y modelo de controladores. También permite abstraer al resto de capas del acceso al hardware.

En el segundo nivel tenemos las **librerías de Android**, estas librerías escritas en C/C++, son utilizadas por varios componentes del sistema y se exponen a los desarrolladores para su uso a través de la capa **Application Framework**.

En el mismo nivel que las librerías se encuentra el **Runtime de Android,** que proporciona un conjunto de librerías base que proporcionan la mayor parte de las funciones disponibles en las bibliotecas base del lenguaje Java. En Android cada aplicación se ejecuta en un proceso distinto en la Máquina virtual (Dalvik o ART). En el caso de Dalvik la aplicación utilizará ficheros .odex que utiliza "just- in-time" (JIT) compilación para compilar el código de bytes cada vez que se inicia una aplicación. En el caso de ART utiliza ficheros .elf que incluyen código nativo (compilado) y permite reducir el tiempo de inicio de la aplicación. En ambos casos, se utilizan ficheros .dex como entrada, que en el caso de Dalvik se transforman en un fichero .odex pasando por un proceso de precompilado y en el caso de ART se generará el fichero .elf. En la siguiente imagen se muestra un poco mejor esta diferencia entre Dalvik y ART:

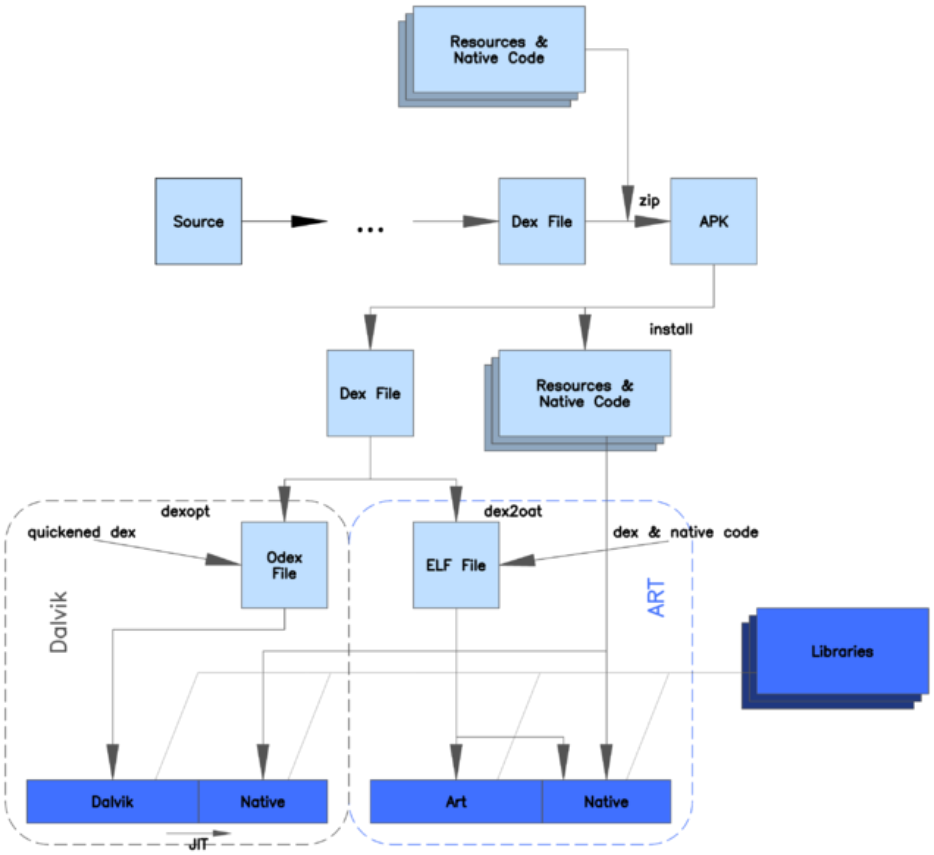


Ilustración 2 - Dalvik vs ART

La capa llamada **Application Framework**, lo que hace es proporcionar a los desarrolladores un conjunto de librerías para que las aplicaciones interactúen con el sistema operativo. Esta capa es la capa de software que proporciona el sistema operativo para, sobre ésta, construir nuestras aplicaciones. Haciendo uso de esta capa, nuestras aplicaciones podrán por ejemplo acceder al Bluetooth del dispositivo, hacer una foto, acceder a la ubicación del dispositivo, ...

En último lugar tenemos el **nivel de Aplicación**, donde se encuentran todas las aplicaciones que nosotros como desarrolladores construimos haciendo uso del interfaz que nos proporciona el nivel inferior.

3.1.3. ¿POR QUÉ UNA APP ANDROID?

Se ha elegido el sistema operativo Android para realizar este proyecto por varias razones.

La principal razón es que Android es claramente el que **mayor cuota de mercado** tiene actualmente y el claro dominante en este aspecto [2, 3]. Según datos de la **International Data Corporation (IDC)**, Android consiguió consolidar durante el año 2014 su posición dominante en el mercado de sistemas operativos para Smartphones, alcanzando un **81,5% de cuota de mercado**, y superando los **mil millones de dispositivos** vendidos. En España, el sistema operativo de Google se encuentra en el **89,4%** de los Smartphones vendidos entre 2014 y 2015.

En la siguiente imagen se puede ver el uso de Android en España frente a otros sistemas operativos móviles [4]:

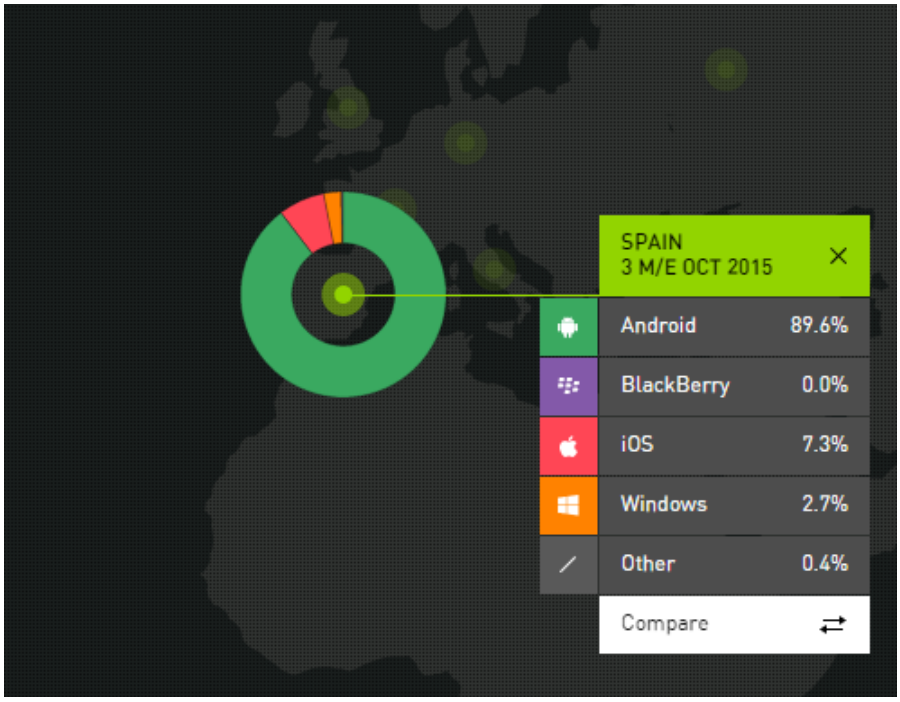


Ilustración 3 - Uso de Android en España

Otra de las razonas es que es el que **mayor comunidad de desarrolladores** tiene y están continuamente desarrollando aplicaciones y librerías para mejorar el ecosistema Android.

Otra razón relacionada con la cuota de mercado es que es un **sistema operativo muy intuitivo y usable,** lo cual permite que un amplio rango de edades sepa utilizarlo con fluidez. En la actualidad, no es nada extraño ver a niños pequeños poniendo música o jugando con un móvil con Android, como tampoco es raro ver a una persona mayor con este tipo de dispositivos.

Como conclusión, los puntos anteriores implican que la elección de Android como sistema operativo móvil para desarrollar esta aplicación, permitirá que la aplicación esté disponible para muchas personas y con un gran abanico de edades.

3.1.4. VERSIONES DE ANDROID SOPORTADAS

La aplicación móvil desarrollada para este proyecto tiene como **API mínima la 21 (Android 5.0)**, por tanto, solo dispositivos con una versión igual o superior a Android 5.0 podrán utilizar esta aplicación.

Esto puede ser un punto negativo, ya que limita el número de usuario que podrá utilizar la aplicación, pero debido a que esta aplicación no está pensada para salir al mercado de manera inmediata, y que el mundo de Android cambia muy rápido y los nuevos móviles que se compran actualmente ya vienen con una versión superior a la 5.0, esto no será un gran problema en el momento que saliese esta aplicación al mercado.

El hecho de limitarnos a esta versión fue por las grandes ventajas que tiene hacer esto y que para poder realizar la solución propuesta teníamos que tener esta versión. Una de las limitaciones es que al usar **Beacons**, estos necesitan que el dispositivo **soporte Bluetooth LE**, que está soportado a partir de la **versión 4.3 de Android** y que en la versión 5.0 han añadido varias mejoras relacionadas con BLE. Otra de las limitaciones está en el uso de **Material Design** que fue presentado con **Android 5.0** y que he querido utilizar para esta aplicación, ya que es el diseño por el que está apostando Google para unificar un poco más el diseño de aplicaciones Android y mejorar así la usabilidad de este sistema operativo.

En definitiva, puede parecer que esto es una limitación, pero en realidad no lo es tanto, ya que las mejoras añadidas en Android 5.0 son bastante importantes como para poder justificar esta decisión, y además, la cuota de mercado que tendríamos actualmente sería de un tercio de los dispositivos Android según la siguiente gráfica[10], y esto seguirá aumentando en el futuro.

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.3%
4.1.x	Jelly Bean	16	8.1%
4.2.x		17	11.0%
4.3		18	3.2%
4.4	KitKat	19	34.3%
5.0	Lollipop	21	16.9%
5.1		22	19.2%
6.0	Marshmallow	23	2.3%

Data collected during a 7-day period ending on March 7, 2016.
Any versions with less than 0.1% distribution are not shown.

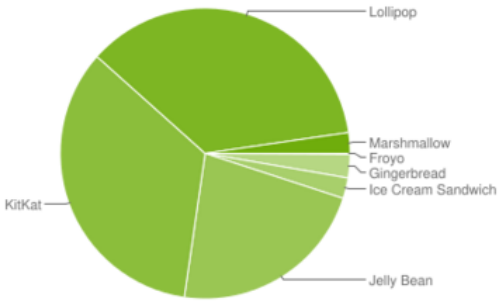


Ilustración 4 - Distribución de versiones Android

[10] Fuente: https://developer.android.com/intl/es/about/dashboards/index.html#Platform

3.2. BLUETOOTH

3.2.1. ¿QUÉ ES BLUETOOTH?

Bluetooth es una especificación que define Redes Inalámbricas de Área Personal (WPAN) que posibilita la transmisión de voz y datos entre diferentes dispositivos mediante un enlace por radiofrecuencia en la banda ISM de los 2,4 GHz [5, 6, 8]. Fue desarrollada por Bluetooth SIG y su funcionamiento está descrito en el estándar **IEEE 802.15.1**.

Los principales **objetivos** que se pretenden conseguir con esta norma son:

- Facilitar las comunicaciones entre equipos móviles.
- Eliminar los cables y conectores entre éstos.
- Ofrecer la posibilidad de crear pequeñas redes inalámbricas y facilitar la sincronización de datos entre equipos personales.

Las **WPAN (redes inalámbricas de área personal),** a cuyo grupo pertenece la tecnología **Bluetooth,** tienen como principal objetivo la transferencia inalámbrica de datos en **cortas distancias** entre un grupo privado de dispositivos. A diferencia de las **WLAN (Redes de área local inalámbrica),** éstas están diseñadas para **no requerir infraestructura, o una infraestructura mínima**.

El objetivo principal de **Bluetooth** es crear **redes Ad-Hoc de bajo coste y consumo. Bluetooth** define un espacio de operación personal **omnidireccional,** permitiendo la **movilidad de los dispositivos**. Existen **tres clases** de dispositivos Bluetooth que se diferencian en su **rango de acción**:

Clase	Potencia máxima permitida (mW)	Potencia máxima permitida (dBm)	Alcance (aproximado)
Clase 1	100 mW	20 dBm	~100 metros
Clase 2	2.5 mW	4 dBm	~5-10 metros
Clase 3	1 mW	0 dBm	~1 metro

Ilustración 5 - Clases de dispositivos Bluetooth

Los dispositivos con Bluetooth también pueden clasificarse según su capacidad de canal:

Versión	Ancho de banda
Versión 1.2	1 Mbit/s
Versión 2.0 + EDR	3 Mbit/s
Versión 3.0 + HS	24 Mbit/s
Versión 4.0	32 Mbit/s

Ilustración 6 - Ancho de banda según versiones de Bluetooth

El estándar Bluetooth define la base del protocolo y la tecnología, lo que ha permitido que surjan multitud de especificaciones que complementan esta definición, este el caso de los **perfiles Bluetooth**. **Existen multitud de perfiles Bluetooth**, cada uno pensado para, a través de la tecnología Bluetooth, definir una especificación que permita ampliar la funcionalidad base.

A continuación una **lista de los principales perfiles Bluetooth** que nos podemos encontrar:

Profile	Description
Generic Access Profile (GAP)	Basic foundation for Bluetooth communications
Service Discovery Application Profile (SDAP)	Allows devices to find each other
Telephone Control Protocol Specification Profile (TCS)	Has group of other profiles and connects Bluetooth devices to one another
Cordless Telephony Profile (CTP)	Connects Bluetooth device to local base station
Intercom Profile (IP)	Connects two or more devices without using a base station
Serial Port Profile (SPP)	Connects Bluetooth devices as serial ports
Headset Profile (HS)	Connects a headset to a phone
Handsfree Profile (HFP)	Used as hands-free car kits
Sim Access Profile (SAP)	Allows carphone to access mobile phone sim
Phone Access Profile (PAP)	Sends information from cars to a mobile phone
Dial-up Networking Profile (DNP)	Connects Bluetooth to a modem for internet browsing
Fax Profile (FP)	Connects Bluetooth to a modem for fax transmission
LAN Access Profile (LAP)	Connects a Bluetooth device to a local area network
Generic Object Exchange	Allows devices to create directories and move
Profile (GOEP)	files across Bluetooth link
Object Push Profile (OPP)	Allows to send a business card or a schedule entry to another Bluetooth device
File Transfer Profile (FTP)	Used to transfer files
Synchronization Profile (SP)	Keeps records between two devices
Basic Printing Profile (BPP)	Sends simple documents to printers
Hard Copy Cable Replacement Profile (HCCR)	Replaces the printer cable
Imaging Profile	Connects image devices such as digital cameras
Human Interface Device Profile (HIDP)	Used for Bluetooth connected keyboards and mice
Local Positioning Profile (LPP)	Connects GPS devices to transfer location data
General Access Video Distribution Profile (GAVDP)	Ideal for systems distributing video and audio streams
Audio Video Remote Control Profile (AVRCP)	Allows Bluetooth devices to control AV equipment
Advanced Audio Distribution Profile (A2DP)	Used for streaming high quality mono or stereo audio
Video Conferencing Profile (VCP)	Allows video conferencing between bluetooth devices
Unrestricted Digital Information Profile (UDIP)	Used for data transfer between 3G phones

Ilustración 7 - Perfiles Bluetooth

Existen varias versiones Bluetooth, aunque en este documento nos vamos a centrar en explicar un poco las novedades de la última versión 4.0. La especificación de **Bluetooth 4.0** incluye la especificación completa del núcleo tanto para el **Bluetooth clásico, Bluetooth de alta velocidad y Bluetooth de bajo consumo (BLE).**

3.2.2. INFORMACIÓN TÉCNICA

La especificación de **Bluetooth** define un canal de comunicación a un **máximo 720 kbit/s** (1 Mbit/s de capacidad bruta) con rango óptimo de 10 m (opcionalmente 100 m con repetidores) [6].

Esta tecnología opera en la **frecuencia de radio de 2,4 a 2,48 GHz** con amplio espectro y saltos de frecuencia con posibilidad de transmitir en **Full Duplex con un máximo de 1600 saltos por segundo**. Los saltos de frecuencia se dan entre un total de **79 frecuencias con intervalos de 1 MHz**; esto permite dar **seguridad y robustez**.

La **potencia de salida** para transmitir **a una distancia máxima de 10** metros es de **0 dBm (1 mW)**, mientras que **la versión de largo alcance transmite entre 20 y 30 dBm (entre 100 mW y 1 W).**

Para lograr alcanzar el objetivo de bajo consumo y bajo costo se ideó una solución que se puede implementar en un solo chip utilizando circuitos CMOS. De esta manera, se logró crear una solución de 9×9 mm y que consume aproximadamente 97% menos energía que un teléfono celular común.

El **protocolo de banda base** (canales simples por línea) **combina conmutación de circuitos y paquetes**. Para asegurar que los paquetes no lleguen fuera de orden, los slots pueden ser reservados por paquetes síncronos, empleando un salto diferente de señal para cada paquete. **La conmutación de circuitos puede ser asíncrona o síncrona**. Cada canal permite soportar tres canales de datos síncronos (voz) o un canal de datos síncrono y otro asíncrono. **Cada canal de voz** puede soportar **una tasa de transferencia de 64 kbit/s en cada sentido**, la cual es suficiente para la transmisión de voz.

Un canal asíncrono puede transmitir como mucho **721 kbit/s en una dirección y 56 kbit/s en la dirección opuesta**. Sin embargo, **una conexión síncrona** puede soportar **432,6 kbit/s en ambas direcciones** si el enlace es simétrico.

3.2.3. ARQUITECTURA HARDWARE

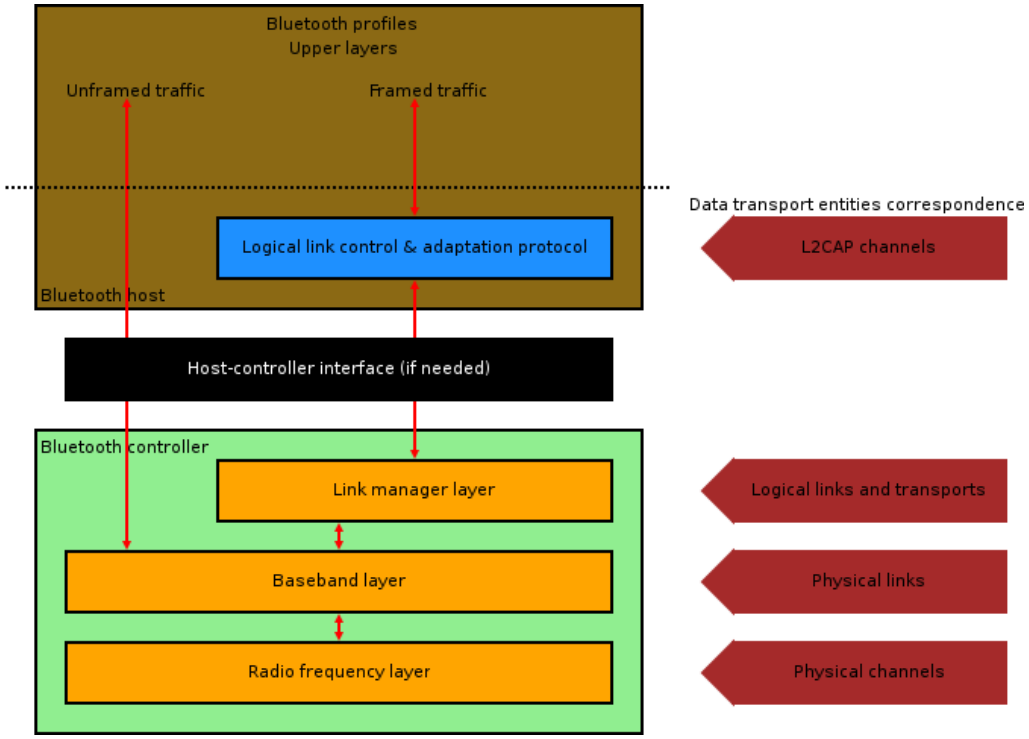


Ilustración 8 - Arquitectura Bluetooth

El hardware que compone el dispositivo Bluetooth está compuesto por dos partes [5]:

- Un **dispositivo de radio**, encargado de **modular y transmitir la señal**. Utiliza la banda ISM de uso no regulado a 2,4 GHz, lo que facilita la consecución de calidad en la señal y la compatibilidad entre transceptores.

- Un **controlador digital**, compuesto por una **CPU**, un **procesador de señales digitales** (DSP - Digital Signal Processor) llamado **Link Controller** (o controlador de Enlace) y de las **interfaces con el dispositivo anfitrión**.

El **LC o Link Controller** se encarga del **procesamiento de la banda base** y del **manejo de los protocolos ARQ y FEC de la capa física**; además, se encarga de las **funciones de transferencia** tanto asíncrona como síncrona, **la codificación de audio y el cifrado de datos**.

La **CPU** del dispositivo se encarga de las instr**ucciones relacionadas con Bluetooth en el dispositivo anfitrión**, para así simplificar su operación.

Para ello, sobre la CPU corre un software denominado **Link Manager** cuya función es la de **comunicarse con otros dispositivos por medio del protocolo LMP**. **El link Manager es el responsable del establecimiento y finalización de conexiones**, así como de su **autenticación** en caso necesario. También realiza el **control del tráfico y la planificación**, junto con la **gestión de consumo y supervisión del enlace.**

3.2.4. BLUETOOTH LOW ENERGY (BLE)

El Bluetooth de baja energía (Bluetooth Low Energy o BLE) fue añadido como parte del estándar 4.0 de Bluetooth [7, 9, 10, 11, 12]. Define una red inalámbrica personal (WPAN) creada con el fin de transmitir datos en **distancias cortas**. Opera en 2.4 GHz con una **tasa de transferencia de 1 Mbps** en la capa física. Es un subconjunto de Bluetooth v4.0 con una **pila de protocolo completamente nueva** para desarrollar rápidamente enlaces sencillos. Está dirigido a aplicaciones de muy baja potencia alimentados con una pila botón.

Las principales diferencias de BLE sobre el Bluetooth tradicional son:

- **Menor consumo** → Puede durar hasta 3 años con la misma pila.
- **Menor coste** → Reduce el coste entre un 60 – 80 % con respecto al Bluetooth tradicional.
- **Utilización** → BLE es recomendable para aplicaciones que requieren pequeñas transferencias de datos periódicas, mientras el Bluetooth tradicional está pensado para comunicaciones más largas y para un gran envío de datos.

La siguiente imagen muestra la pila del protocolo Bluetooth en sus tres posibles modalidades, donde podemos ver que los cambios de la pila definida por Bluetooth LE y el Bluetooth tradicional:

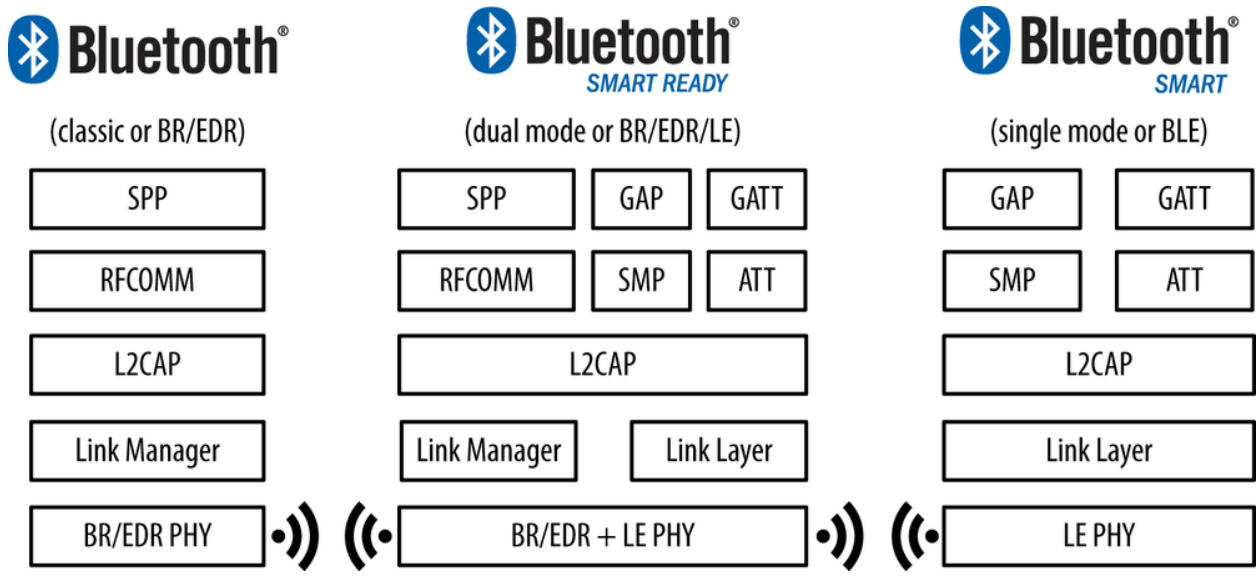


Ilustración 9 - Pila de protocolo Bluetooth

De la anterior pila de protocolos se pueden destacar los siguientes elementos:

- **GAP** → Se encarga de **controlar las conexiones y la publicación de información** del dispositivo Bluetooth LE. Hace que el dispositivo **sea visible** y determina cómo dos dispositivos pueden o no interactuar entre sí.

- **GATT** → Define cómo dos dispositivos Bluetooth LE transfieren información.

- **ATT** → La transferencia de información en dispositivos Bluetooth LE se hace mediante el protocolo **ATT** (Attribute Protocol)

- **SMP** → Protocolo encargado de **controlar la seguridad en las conexiones**. Para ello utiliza servicios de **Autenticación y autorización de dispositivos**, así como servicios encargados de asegurar la **integridad, confidencialidad y privacidad de los datos**.

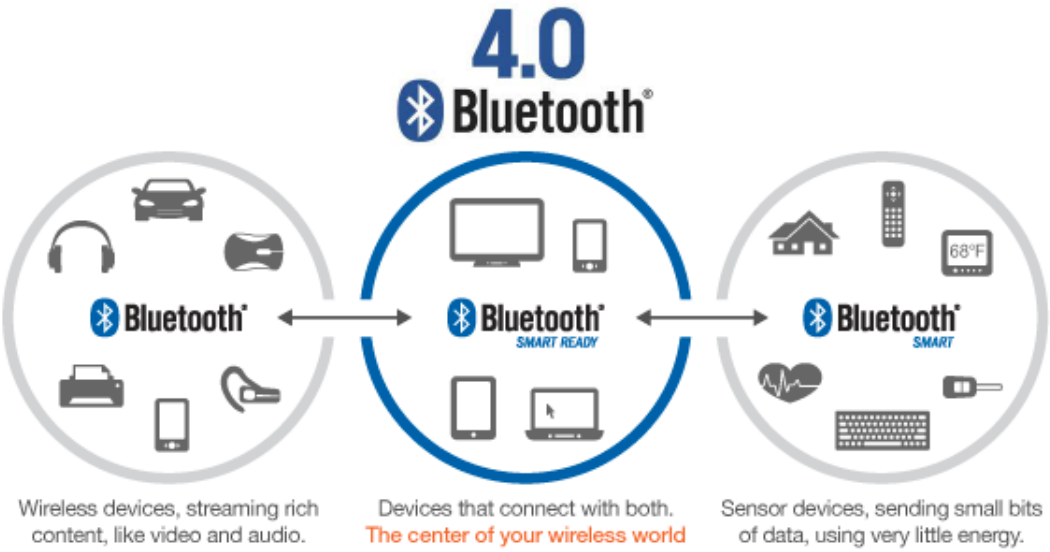


Ilustración 10 - Bluetooth 4.0

Bluetooth SMART tiene como objetivo **clarificar la compatibilidad entre dispositivos Bluetooth LE**. Existen dos nomenclaturas dentro de Bluetooth Smart:

- **Bluetooth SMART Ready** → indica un dispositivo que funciona en **modo dual**, es decir, que opera tanto con dispositivos **Bluetooth Classic**, como con dispositivos Bluetooth LE. Suelen ser la mayoría de dispositivos modernos como Smartphones, Tablets, Portátiles…

Ilustración 11 - Bluetooth Smart Ready

• **Bluetooth SMART** → indica un dispositivo que solo soporta Bluetooth LE y requiere de un dispositivo Bluetooth SMART Ready o Bluetooth SMART para funcionar. Normalmente son pequeños dispositivos con aplicaciones concretas como pulsómetros, relojes, Beacon, etc…

Ilustración 12 - Beacon

Nota → En el caso de dispositivos **Smart Ready,** hay que tener en cuenta que el **Sistema Operativo** también tiene que ser compatible con **Bluetooth LE**. En **Android** está disponible desde la **versión 4.3,** mientras que en **IOS desde** la versión de **IOS 5**.

3.3. BEACONS

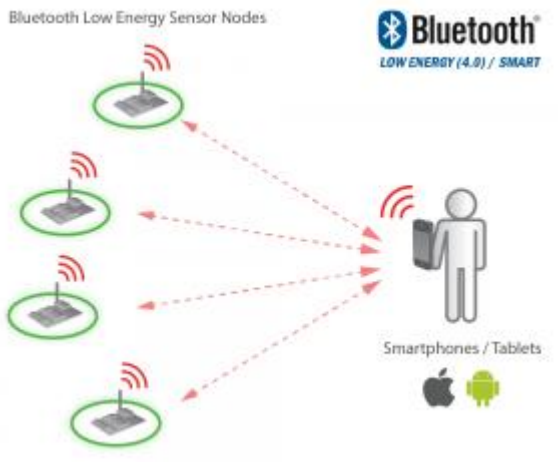
3.3.1. ¿QUÉ ES UN BEACON?

Un **Beacon o iBeacon (nombre que le puso Apple)** es un pequeño **dispositivo Bluetooth Smart que se encarga de emitir una señal en el rango de alcance que tiene cualquier dispositivo Bluetooth Low Energy (BLE) [13, 14, 15, 16, 17].** Este dispositivo **emite pequeñas piezas de información** con cierta frecuencia. La información que puede emitir un dispositivo Beacon puede ser: información ambiental (temperatura, presión del aire, humedad), información de localización u orientación, ...

La información que envía un Beacon normalmente es estática, aunque puede cambiar con el tiempo. Debido al uso de BLE los Beacons pueden funcionar durante años con una simple pila de botón, lo que lo convierte en un dispositivo perfecto para lo que necesitamos.

Por tanto, una cosa que debe quedar clara es que con estos dispositivos no se puede interactuar, simplemente están emitiendo información continuamente y son los dispositivos BLE que estén cerca de éstos los que pueden consultar esta información e interpretarla.

3.3.2. EXPLICACIÓN DEL FUNCIONAMIENTO



Como se ha expuesto en el punto anterior, un Beacon emite información de forma periódica. La siguiente imagen muestra la estructura de una **PDU (Unidad de datos del Protocolo) de BLE** y dónde se localiza dentro de esta PDU la información relativa al Beacon:

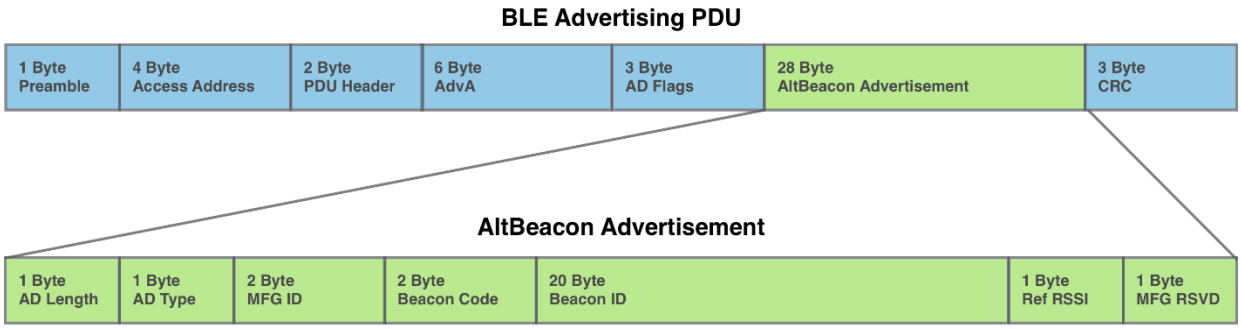


Ilustración 13 - Unidad de datos del protocolo BLE

En la siguiente imagen podemos ver mejor la distribución de la información que emite el Beacon:

Offsets	Octet	0	1	2	3
Octet	Bit	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
0	0	AD LENGTH	AD TYPE	MFG ID	
4	32	BEACON CODE		BEACON ID	
8	64	BEACON ID (CONTINUED)			
12	96	BEACON ID (CONTINUED)			
16	128	BEACON ID (CONTINUED)			
20	160	BEACON ID (CONTINUED)			
24	192	BEACON ID (CONTINUED)		REFERENCE RSSI	MFG RESERVED

Ilustración 14 - Distribución de la información de un Beacon

De la información anterior las partes más importantes para el proyecto son:

- **BEACON ID (20 bytes)** → 16 bytes para el **UUID**, 2 bytes para el **Major** y 2 bytes para el **Minor**. Como vemos, el identificador de un Beacon está compuesto de tres elementos, el **UUID** que es un **identificador único universal**, luego están el **Major** y el **Minor**. El **Major** sería como un identificador de grupo, mientras el **Minor** sería un identificador de un dispositivo concreto dentro de ese grupo. En nuestra aplicación, **el UUID de nuestros Beacons será siempre el mismo, el Major identificará la tienda y el Minor el Beacon concreto dentro de esa tienda**.

- **RSSI** → Este campo nos indica la potencia de la señal recibida. Se utiliza para **calcular la distancia** a la que se encuentra el Beacon.

Nota → Esta información está sacada de la documentación de la especificación **AltBeacon** por lo que la longitud de los campos cambia con respecto a los de iBeacon, ya que no hay definido un estándar único para estos dispositivos.

En este proyecto se están utilizando con dos propósitos que son **Proximity marketing** y **Check-in coupons**. Para hacer esto, simplemente tenemos un Beacon con **dos rangos de acción**, cuando el usuario está fuera de la tienda se le incita a entrar (**Proximity marketing**) y cuando el usuario está dentro se le puede dar la bienvenida y un código de descuento para su compra (**Check-in coupons**). En la siguiente imagen se puede ver estas dos características que ofrecen los Beacons:

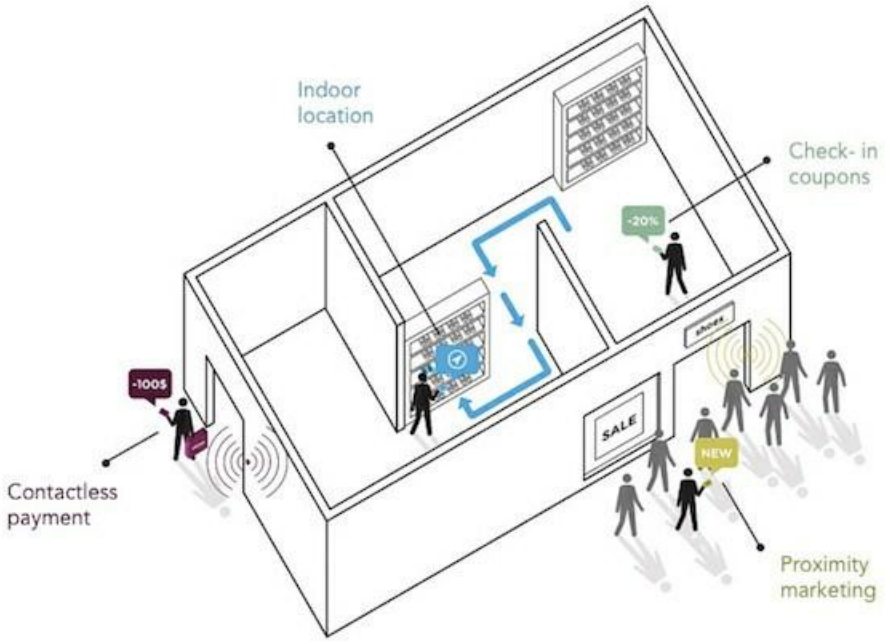


Ilustración 15 - Ejemplo de uso de Beacons

En la imagen anterior también aparecen otras dos posibilidades de uso de estos dispositivos. El **location indoor** que permitiría localizar al usuario dentro de la tienda, no es algo que interese en este proyecto, aunque sí se podrían utilizar los Beacons para dar información acerca de ciertos productos o secciones del establecimiento al consumidor. La otra característica llamada **Contactless payment** no creo que se pueda realizar con Beacons, y para ello se ha pensado en utilizar en su lugar **tecnología NFC**, que podría utilizarse para hacer compatible la aplicación a los nuevos sistemas de pago que están emergiendo ahora como **Android Pay** y **Apple Pay**.

3.3.3. ERRORES COMUNES QUE SE COMETEN AL HABLAR DE BEACONS

A continuación se va a recopilar una pequeña lista de errores comunes que se cometen en algunos artículos a la hora de hablar de dispositivos **Beacons**.

- **No puede sustituir a NFC para pagos** ya que tiene menos seguridad y lo más importante, solo emite información, no permite recibir y tampoco están conectados a internet ni nada por el estilo, por lo cual no se me ocurre forma de realizar pagos utilizando esta tecnología.

- **No pueden emitir un mensaje concreto a un usuario,** sino que es la aplicación que escucha los Beacons la que tiene que decidir en función de la información que transmite el Beacon qué mensaje le tiene que mostrar al usuario logueado.

- **La distancia a la que pueden funcionar no es de 50m**. Esa sería la distancia máxima teórica que en la realidad nunca se logra conseguir debido a que BLE trabaja en la misma banda de frecuencia que los dispositivos WIFI y, por tanto, surgen muchas interferencias que impiden que se alcance ese límite teórico. En las pruebas que yo he hecho en un entorno real, el dispositivo Beacon logro alcanzar la distancia de 15m, por lo que yo diría que sobre 20m podría ser el máximo que alcanzaría si se configurase el Beacon para que emitiese con más potencia (cosa que sería contraproducente porque haría que se redujese su autonomía)

- **No permiten localizar directamente a un usuario** sino que **determinan su distancia <u>aproximada</u> al Beacon**. Recalco lo de aproximada porque como he comentado en el punto anterior, uno de los problemas conocidos de los dispositivos BLE es que trabajan en la misma frecuencia que WIFI y esto hace que tengan muchas interferencias y, por tanto, el margen de error de la distancia es bastante grande y varía según las interferencias del entorno. Esta tecnología no está pensada para ubicar a una persona dentro de un establecimiento, aunque sí se podría lograr pero se necesitarían varios dispositivos (más de tres) para poder triangular de manera correcta la ubicación del usuario.

- **No se activan cuando llega un receptor,** sino que están continuamente enviando información (cada cierto intervalo de tiempo establecido en la configuración).

3.3.4. APORTE AL PROYECTO

Los **Beacons** juegan un papel fundamental en el proyecto. Su función es la de proporcionar contexto a la aplicación para saber la zona por la que se encuentra el usuario y qué información puede ser la que más le interese.

El papel de los **Beacons** en el proyecto es simplemente el comentado anteriormente, dar contexto a la aplicación para saber dónde se encuentra el usuario, y será **la aplicación la que actuará según ciertos parámetros,** por ejemplo, **la distancia al Beacon o el Beacon concreto que se ha detectado, ya que cada Beacon estará asociado a una tienda y a una posición dentro de ésta y tendrán asociados ciertos anuncios que mostrarán al usuario.**

3.4. NFC

3.4.1. ¿QUÉ ES NFC?

NFC (Near Field Communication) es una **tecnología de comunicación inalámbrica**, de corto alcance (funciona por proximidad) y alta frecuencia que permite el intercambio de datos entre dispositivos [18, 19, 20, 21]. Fue aprobada como estándar ISO/IEC en 2003.

NFC se comunica mediante **inducción en un campo magnético**, donde dos antenas de espiral son colocadas dentro de sus respectivos campos cercanos. Se trata de una tecnología inalámbrica que funciona en la **banda de los 13.56 MHz** (esta banda no necesita licencia para usarla) y que deriva de las etiquetas RFID. Su **tasa de transferencia puede alcanzar los 424 kbit/s** por lo que su enfoque más que para la transmisión de grandes cantidades de datos es para comunicación instantánea, es decir, identificación y validación de equipos/personas.

Su punto fuerte está en la **velocidad de comunicación**, que es **casi instantánea sin necesidad de emparejamiento previo**. A su favor también juega que su uso es transparente a los usuarios, y que los equipos con tecnología **NFC** son capaces de **enviar y recibir información al mismo tiempo**.

El **alcance** de la tecnología NFC es **muy reducido**, pues se mueve como máximo **en un rango de los 20 cm**, y aunque esto podría considerarse un punto negativo, en realidad es un punto positivo ya que implica **mayor seguridad** y esta es una de las principales razones por las que se ha elegido esta tecnología para realizar pagos en un futuro. Aunque es verdad que por sí solo esta característica de NFC no hacen seguras sus comunicaciones y hay que emplear mecanismos para garantizar la confidencialidad, integridad y autenticación de los datos transmitidos, por ejemplo utilizando **SSL**.

La tecnología NFC soporta **dos modos de funcionamiento**:

- **Activo** → en el que ambos equipos con chip NFC generan un campo electromagnético e intercambian datos (caso de los Smartphones).

- **Pasivo** → en el que solo hay un dispositivo activo y el otro se aprovecha de la modulación de la carga para poder transferir los datos (caso de las pegatinas). El iniciador de la comunicación es el encargado de generar el campo electromagnético.

3.4.2. ARQUITECTURA

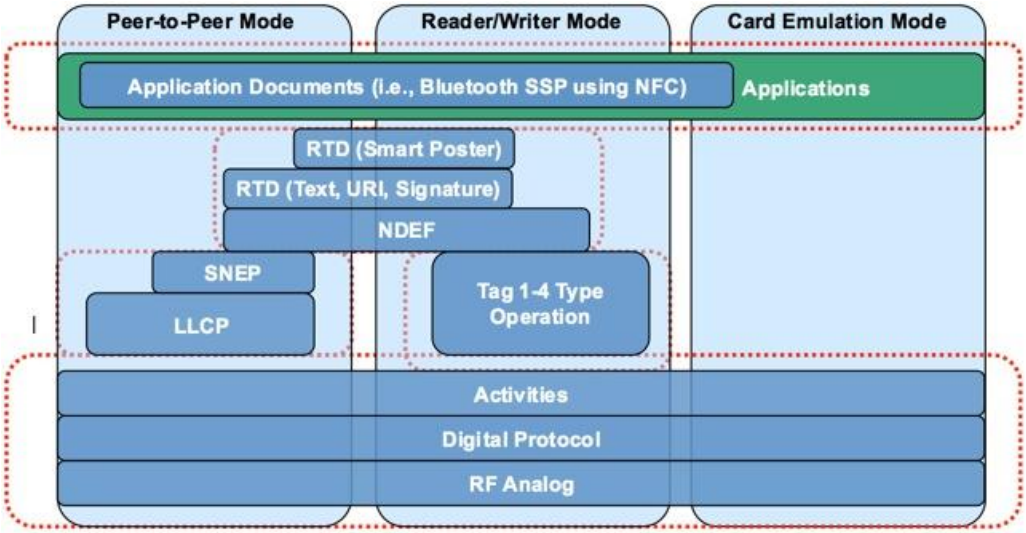


Ilustración 16 - Arquitectura NFC

La **arquitectura NFC** se basa principalmente en los siguientes componentes [22, 23]:

- **RTD (Record Type Definition)** → Define los **tipos de registros** que se pueden enviar a través de NFC. Estos tipos son: MIME, URI, RTD. Existen **dos tipos de RTDs**:
    - **Well-Known types**: que se utiliza cuando no hay un MIME o URI equivalente (Text, Generic control, Smart poster, Signature)
    - **External types**: es el que utilizan las empresas para definir su propio espacio de nombres.
- **NDEF (NFC Data Exchange Format)** → Define el intercambio de información entre dos dispositivos (P2P, Reader/Writer). La información se manda en mensajes que están compuestos de varios registros. Cada registro contiene longitud, tipo, identificador y payload.
- **SNEP (Simple NDEF Exchange Protocol)** → Protocolo de petición/respuesta. Define cómo se realiza el intercambio de mensajes NDEF.

- **NFC Tags Types →** En la actualidad existen **4 tipos de etiquetas NFC**. Estas etiquetas definen una serie de características.

    - **Tag type 1 →** Esta tag es de lectura/escritura (se puede configurar de solo lectura). Memoria de 96 bytes- 2 Kbytes. Velocidad 106 Kbits/s. No soporta anticolisión. Ejemplo: Topaz.

    - **Tag type 2 →** Esta tag es de lectura/escritura (se puede configurar de solo lectura). Memoria de 96 bytes- 2 Kbytes. Velocidad 106 Kbits/s. Soporta anticolisión. Ejemplo: NXP Mifare Ultralight.

    - **Tag type 3 →** Esta tag solo permite lectura y/o escritura (Configurado de fábrica). Memoria de hasta 1Mb. Velocidad 212 o 424 Kbits/s. Soporta anticolisión. Ejemplo: Sony-Felica.

    - **Tag type 4 →** Esta tag solo permite lectura y/o escritura (Configurado de fábrica). Memoria de hasta 32Kb. Velocidad 106, 212 o 424 Kbits/s. Soporta anticolisión. Ejemplo: NXP Desfire.

    - **Mifare Classic →** No son estándar. Soportados por móviles con chip NXP. Esta tag es de lectura/escritura (se puede configurar de solo lectura). Memoria de 192, 768 o 3584 bytes. Velocidad 106 Kbits/s. Soporta anticolisión. Ejemplo: NXP Mifare Classic.

- **LLCP (NFC Logical Link Control Protocol) →** Define un protocolo de comunicación para comunicaciones P2P en el nivel de enlace. Esta especificación define **dos tipos de servicios: sin conexión y orientado a conexión,** organizados en tres clases de servicios de enlace: sin conexión, orientado a conexión y ambos. El servicio sin conexión ofrece una configuración mínima que no garantiza la fiabilidad ni el control de flujo. En el servicio orientado a conexión, se garantiza el control de flujo, orden, entrega confiable y multiplexado basado en sesiones.

3.4.3. USOS DE LA TECNOLOGÍA NFC

Los principales usos que se le pueden dar a esta tecnología son:

- **Identificación** → Permite la transferencia de información de identificación a un dispositivo que realice las funciones de control de acceso.

- **Intercambio de información** → Permite el intercambio de información entre dos dispositivos NFC.

- **Pagos** → Como ya he comentado en varias ocasiones, la tecnología NFC será en un futuro utilizada para permitir los pagos a través de dispositivos móviles, mejorando mucho la experiencia de los consumidores al realizar los pagos con un sistema de **Contactless**.

En este proyecto se está utilizando para el intercambio de información de identificación del cliente, aunque en un futuro se podría utilizar también para pagos y para dar más información sobre un producto concreto a través de las pegatinas NFC.

3.4.4. ¿PODRÍA SUSTITUIR A LOS BEACONS?

Una pregunta que podría surgir es si esta tecnología podría haber sustituido a los Beacons, y la respuesta es sí, aunque la experiencia no sería la misma. Se podrían sustituir los Beacons por etiquetas NFC y que el usuario pasase su móvil por delante de ellas para que la aplicación actuase igual que al detectar un Beacon, pero esto tendría varios inconvenientes:

- La **experiencia de usuario** sería mucho peor, ya que sería el usuario el que tendría que, activamente, acercarse a una de estas etiquetas NFC y poder así obtener la información. Esto podría crear colas en las entradas o frente a los productos. Sin embargo, utilizando Beacons no es necesario que el usuario solicite activamente la información, sino que la aplicación va a detectar el Beacon y le mostrará la información asociada sin que el usuario tenga que preocuparse de buscarlo.

- Dificultaría el **llamar la atención a nuevos clientes**, ya que si se utilizasen etiquetas NFC solo se podría llegar a usuarios interesados y no llamar la atención de usuarios que tienen la aplicación instalada y pasan por delante de una tienda que, hasta el momento, nunca le ha llamado la atención.

- Para evitar colas en caso de que se utilizase NFC, habría que poner muchas etiquetas NFC que tengan el mismo propósito, por ejemplo, varias etiquetas NFC para el propósito de **Check-in coupons.**

En resumen, podrían sustituir a los Beacons, pero no tendrían las mismas posibilidades principalmente porque NFC es una tecnología de corto alcance y, por tanto, no sería la tecnología más adecuada para nuestro propósito de obtener información contextual de los clientes y proporcionarles información según estos datos.

3.4.5. APORTE AL PROYECTO

Se ha utilizado principalmente para la identificación de los clientes y así poder registrar la compra del cliente y contabilizar sus puntos. En un futuro se podría utilizar para realizar pagos a través de esta tecnología, utilizando el sistema de pago que propone Google llamado Android Pay. También se podría utilizar esta tecnología para poder dar más información sobre un producto en la aplicación mediante el uso de pegatinas NFC.

En la aplicación, el usuario tendrá la opción de enviar por NFC su código de cliente al dependiente que le esté atendiendo, facilitando así la identificación del cliente.

3.5. CÓDIGOS QR

3.5.1. ¿QUÉ SON LOS CÓDIGOS QR?

Un **Código QR (Quick Response)** es un código de barras bidimensional que permite representar una información que puede leer un intérprete de códigos QR [25]. Estos códigos fueron creados en 1994 por la compañía Japonesa Denso Wave. En el año 1999 fue lanzado al público bajo licencia libre y en el año 2000 los códigos QR fueron estandarizados (ISO/IEC18004).

En la actualidad, los códigos QR se utilizan como un mecanismo para compartir información de forma rápida a dispositivos móviles, eliminando la necesidad de que el usuario tenga que introducir cierta información de forma manual en sus dispositivos, por ejemplo una URL de una web.

3.5.2. EXPLICACIÓN DEL FUNCIONAMIENTO

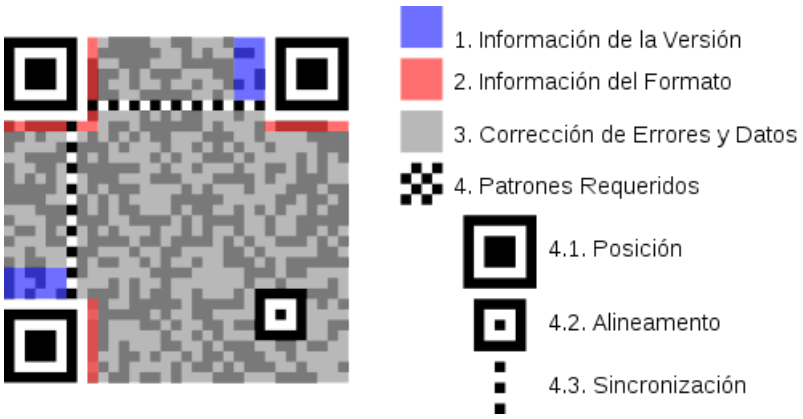


Ilustración 17 - Formato Código QR

Dentro de un código QR se pueden diferenciar diferentes partes, como se muestra en la imagen anterior. El lector recogerá toda la información que proporciona el código QR teniendo en cuenta **la posición, el alineamiento y la sincronización [24]**.

Una de las mejores características de estos códigos es que **son neutrales respecto a la orientación**, es decir, no es necesario estar exactamente frente a ellos o alineados verticalmente para escanearlos. La imagen en sí proporciona toda la información necesaria para permitir al software girarla, orientarla y aplanarla, incluso si se fotografía el código con ángulo.

La característica más prominente del código QR son los **tres cuadrados que aparecen en tres de las cuatro esquinas de la imagen**. Son tres porque **facilitan una orientación rotacional rápida y a la vez proporcionan un inmediato sentido del tamaño y orientación angular**. El hecho de que exista un cuadrado grande en la esquina inferior derecha aporta una idea instantánea de orientación rotacional. En la siguiente imagen se puede ver que el cuadrado más pequeño está a 4 puntos desde la base de la imagen y 4 puntos desde la derecha:

Ilustración 18 - Marca de sincronización en Códigos QR

Hay algo que se encuentra en todos y cada uno de los QR que existen, y es una **marca de sincronización** que une las esquinas interiores de los cuadrados grandes. Mirando entre los cuadrados superiores, se aprecia que siempre hay la siguiente sucesión (negro/blanco, negro/blanco). Este diseño permite tener una referencia del tamaño y de nuevo orientación posicional adicional.

Ilustración 19 - Número de formato y versión Código QR

El código en sí tiene un **número de formato y de versión** almacenado en los puntos circundantes a los tres cuadrados grandes (ver imagen inicial).

Uno de los problemas que se encontraron los diseñadores fue a la hora de que los datos emularan las **características de reparación del código QR**. Habría un **problema de señalización en banda**, es decir, cómo separaban la información relativa al formato del contenido. **El código QR actúa de forma similar a los discos duros** usando una tecnología llamada Self-Clocking (Auto-Sincronizable). En lugar de malgastar espacio con una señal de reloj o una traza de reloj además de los datos, **organizaron los datos de forma que fueran auto-sincronizables**, para que proporcionen su propia información de sincronización.

El modo de hacerlo es asegurarse que no hay, en el caso de los discos duros, un grupo de ceros juntos. Tener esta sucesión de ceros juntos significa que no está pasando nada, el problema surge cuando pasa algo (aparecen 1s) es necesario saber exactamente cuántas cosas no pasaron, es decir, cuantos ceros había. Y puede ser peligroso si el disco duro no gira a ritmo constante. De forma similar, si el código QR está estirado o arrugado causará un cambio local en la frecuencia del patrón establecido en el campo visual. Así que **no es buena idea tener grandes bloques negros o blancos en el código** porque sería un problema a la hora de conocer su tamaño, lo cual es crucial, especialmente en códigos de gran densidad que se lean desde cierta distancia.

La **solución** tomada por los diseñadores fue establecer una **máscara para uno de los formatos de control y realizar un XOR a todo el conjunto de datos**. La operación XOR se usa muy a menudo en criptografía. **Al realizar un XOR a algún dato, se invierten los bits, si se vuelve a realizar un XOR sobre los mismos datos, se obtiene el dato original**. Debido a lo simple de realizar un XOR es el proceso perfecto para resolver el problema. Dependiendo de la naturaleza de los datos contenidos en el código QR crearon una librería con ocho patrones XOR distintos derivados matemáticamente de las coordenadas X e Y de una porción de 8×8.

Estos patrones de 8×8, por ejemplo, son un tablero de ajedrez, otro son rayas verticales, rayas horizontales etc. Llegados a este punto, la tarea del codificador QR es establecer un código QR simple, sin enmascarar ni XOR y luego, seguir un criterio para detectar posibles problemas, como grandes secuencias de ceros o unos, o formaciones de grandes bloques**. Lo que se hace es aplicar cada una de las ocho máscaras para obtener ocho posibles códigos QR candidatos y basándose en un criterio, se elige el más adecuado, y almacenará la máscara elegida como información de codificación.**

Finalmente para **decodificar el contenido,** se empieza por la esquina inferior derecha, donde se encuentra el tipo de codificación (de 4 puntos de tamaño), luego la longitud (8 puntos) seguido de los datos almacenados. Después de los datos puede haber otra codificación y otra longitud para seguir obteniendo datos. Por consiguiente, es posible disponer de múltiples formatos en un solo código QR. Incluso densidad y corrección de errores variable. Hay que tener una concepto claro, **la corrección de errores significa redundancia,** a más corrección de errores, mayor porcentaje del área no serán datos.

CAPACIDAD DE ALMACENAMIENTO

Solo numérico	**Máx. 7.089 caracteres**
Alfanumérico	Máx. 4.296 caracteres
Binario	Máx. 2.953 bytes
Kanji/Kana	Máx. 1.817 caracteres

Tabla 1 - Capacidad de almacenamiento en Código QR

Nivel L	**7% de las claves se pueden restaurar**
Nivel M	15 % de las claves se pueden restaurar
Nivel Q	25 % de las claves se pueden restaurar
Nivel H	30 % de las claves se pueden restaurar

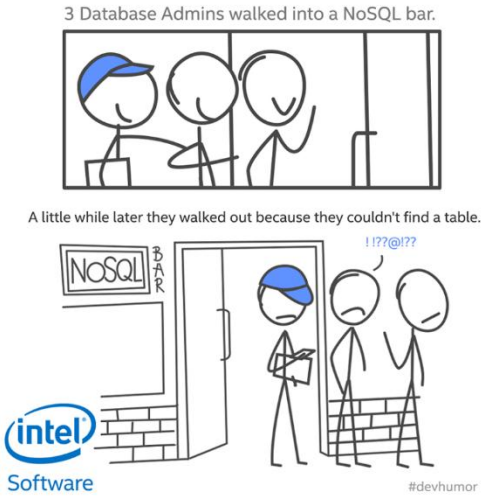
Tabla 2 - Capacidad de corrección de errores en Código QR

3.5.3. APORTE AL PROYECTO

Los códigos QR, al igual que el NFC, es una **alternativa para facilitar la identificación del cliente** frente al establecimiento. El cliente dispondrá en la aplicación de una opción para generar un código QR que represente su código de cliente, y que podrá ser leído posteriormente por el dependiente del establecimiento para asociarle una compra y facilitar así la identificación del cliente.

También se ha pensado que estaría bien que los clientes tuviesen una tarjeta con este código QR, por si algún día no disponen del móvil, puedan igualmente asociar las compras a su cuenta utilizando esa tarjeta.

3.6. BBDD NOSQL



3.6.1. ¿QUÉ SON LAS BBDD NOSQL?

Tradicionalmente se vienen utilizando sistemas de gestión de bases de datos relacionales (RDBMS) para almacenar la información en las organizaciones. Su éxito se basó en que son una solución para los problemas de gestión y estructuración de la información de las organizaciones, con un fundamento matemático muy fuerte, lenguaje estandarizado (aceptado y adoptado) para su gestión (SQL), con metodologías estructuradas formales para el diseño de los sistemas de información de las organizaciones y con principios de diseño como la regla **ACID** (**Atomicidad, Consistencia, Disponibilidad (Availability) y Durabilidad**). Las bases de datos relacionales más conocidas son: Oracle, MySQL, SQL Server, PostgreSQL, DB2, …

NOSQL (Not Only SQL) es una amplia clase de **sistemas de gestión de bases de datos** que difieren del modelo clásico del sistema de gestión de bases de datos relacionales (RDBMS) en aspectos importantes, como por ejemplo: **los datos almacenados no requieren estructuras fijas** como tablas (no tienen esquema), normalmente **no soportan operaciones JOIN**, **no garantizan completamente ACID** (atomicidad, consistencia, aislamiento y durabilidad), y habitualmente **escalan bien horizontalmente**, resuelven el problema de los altos volúmenes de información y la inmensa cantidad de consultas y transacciones diarias [27, 28, 30]. Los sistemas NOSQL se denominan a veces "no solo SQL" para subrayar el hecho de que también pueden soportar lenguajes de consulta de tipo SQL.

NOSQL VS SQL SUMMARY

	SQL DATABASES	NOSQL DATABASES
Types	One type - minor variations	Many different types
Development History	1970s	2000s
Examples	MySQL, Postgres, Oracle Database	MongoDB, Cassandra, HBase, Neo4j
Data Storage Model	Relational	Varies based on database type.
Schemas	Structure and data types are fixed in advance.	Typically dynamic.
Scaling	Vertically,	Horizontally,
Development Model	Mix	Open-source
Supports Transactions	ACID	In certain circumstances and at certain levels (e.g., document level vs. database level)
Data Manipulation	SQL	Through object-oriented APIs
Consistency	Can be configured for strong consistency	Depends on product.

Ilustración 20 - NoSQL vs SQL

EL PROBLEMA QUE BUSCAN SOLUCIONAR

Pero la gran pregunta podría ser: ¿Qué problema se busca solucionar con las bases de datos NOSQL? Con los siguientes datos quizás se pueda entender el problema:

- Desde 2010 se están vendiendo más dispositivos móviles que PCs.
- Son más de 900 millones los usuarios de Facebook.
- Cada minuto se generan 50 horas de contenido en YouTube
- Twitter genera casi 8 terabytes de datos con sus más de 90 millones de tuits al día.
- Wall-Mart gestiona un millón de transacciones de sus clientes/ hora (2.5 petabytes)

Se han creado más datos en los últimos dos años que todos los años anteriores, se han creado datos del orden de ExaBytes (10^{18}) por año. Los datos son más entrelazados y conectados, son datos menos estructurados y datos a escala de la web, con mucha lectura/escritura, los esquemas cambian frecuentemente, por ejemplo las aplicaciones sociales no necesitan el mismo nivel de ACID y la orientación del software es hacia servicios (PaaS: programas como Servicios)

El problema aparece con los sistemas de millones de transacciones al día contra la base de datos, otro elemento más es que se necesita cada vez mayor flexibilidad para escalar (escalabilidad) y porque para solucionarlo se estaban adquiriendo mayores y más potentes computadores. Por tanto, las bases de datos NOSQL intentan resolver problemas de almacenamiento masivo, alto desempeño, procesamiento masivo de transacciones (sitios con alto transito) y, en términos generales, ser alternativas NOSQL a problemas de persistencia y almacenamiento masivo (voluminoso) de información para las organizaciones.

Pero la gran diferencia es cómo almacenan los datos. Por ejemplo, una factura en el modelo relacional termina guardándose en 4 tablas (con 3 o 4 claves foráneas – asociaciones involucradas) y en NOSQL simplemente guardan la factura y no se diseña las tablas ni su estructura por adelantado, se almacena, por ejemplo, una clave (número de la factura) y el Objeto (la factura). Unido a lo anterior podemos afirmar que **en las bases de datos relacionales: la lectura de datos es muy costosa**, existe **mucha transaccionalidad innecesaria**, se asume que los datos son densos y bien estructurados, tienen **problema de escalabilidad horizontal** y no todos los problemas se pueden modelar para una base un RDBMS.

3.6.2. TEOREMA CAP

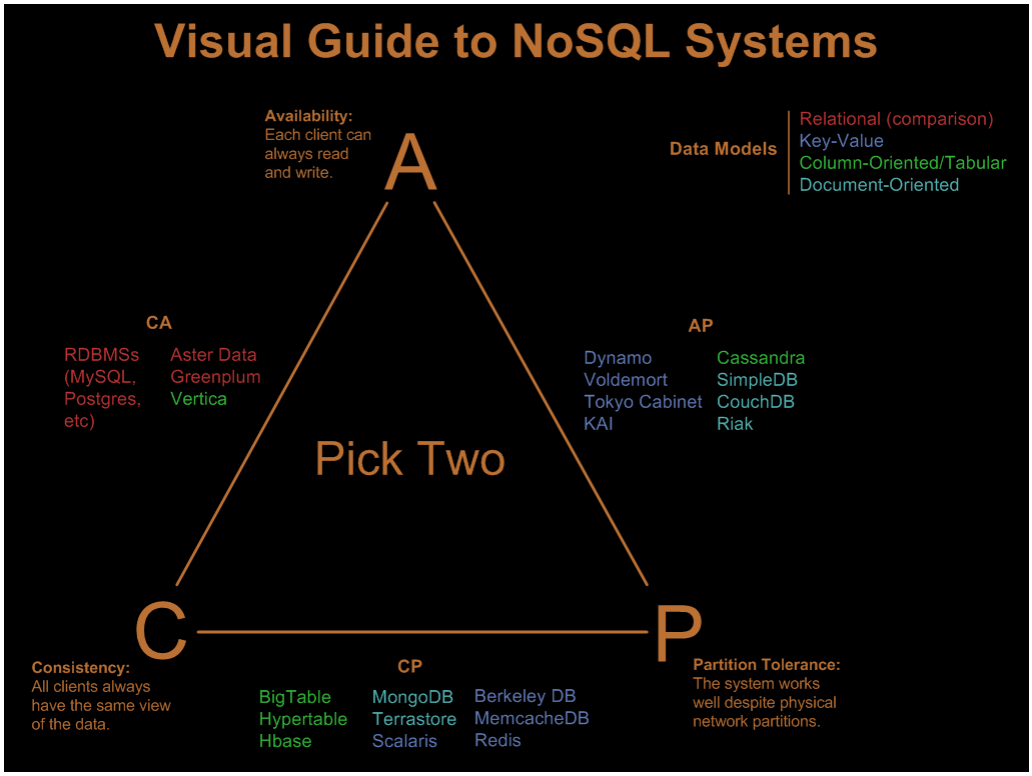


Ilustración 21 - Clasificación de Bases de datos según el teorema de CAP

Según el **teorema de CAP** definido por **Erik Brewer** (año 2000), las bases de datos solo pueden garantizar dos de estas tres características [26, 27]:

- **Consistencia** → Que todos los nodos vean la misma información al mismo tiempo.
- **Disponibilidad ("Availability")** → Garantía de que cada petición a un nodo reciba una confirmación de si ha sido o no resuelta satisfactoriamente.
- **Tolerancia a particiones** → El sistema sigue funcionado a menos que haya un fallo total de red. Es decir, que si afecta solo a unos nodos, el sistema debe seguir funcionando.

Como se puede ver en la imagen del principio, **las bases de datos relacionales se centran más en asegurar la Consistencia y Disponibilidad, dejando un poco de lado la Tolerancia a particiones**. En la imagen también se puede ver como las **bases de datos NoSQL** le **dan más importancia a la Tolerancia a particiones**, existiendo bases de datos NoSQL que también buscan ofrecer Consistencia, como por ejemplo MongoDB, y otras que buscan ofrecer Disponibilidad como puede ser Cassandra.

3.6.3. BASE (BASICALLY AVAILABLE, SOFT STATE, EVENTUAL CONSISTENCY)

BASE fue definida por **Eric Brewer**, quien también es conocido por la formulación del teorema de CAP [27, 31, 32]. Como hemos visto en el punto anterior, el teorema CAP define tres características que puede tener un sistema (Consistencia, disponibilidad y tolerancia a particiones) de las cuales solo podrá asegurar dos de ellas simultáneamente.

ACID define las características que tienen que cumplir las transacciones en los sistemas de bases de datos SQL (**Atomicidad, Consistencia, Disponibilidad (Availability) y Durabilidad**). Es decir, se centra en la consistencia y disponibilidad del sistema CAP.

BASE se utiliza para describir las características de los sistemas NOSQL:

- **Basically Available** → Indica que el sistema garantiza la disponibilidad, incluso en la presencia de múltiples fallos. Esto se consigue con la replicación de los datos y con esto se garantiza la disponibilidad y tolerancia a particiones del teorema CAP.

- **Soft state** → Indica que el estado del sistema puede cambiar con el tiempo, incluso sin entrada. Es decir, no se asegura la consistencia del sistema.

- **Eventual consistency** → Indica que el sistema será consistente en un punto del futuro. Es decir, los datos convergerán a ser consistentes en el futuro. En ACID, la consistencia tiene que ser inmediata, sin embargo en BASE no se toma tan crítico que el sistema no sea consistente en un momento dado.

BASE se centra en asegurar la **Tolerancia a particiones y disponibilidad** del sistema según el teorema CAP, **dejando un poco de lado la Consistencia**.

3.6.4. CLASIFICACIÓN DE BBDD NOSQL

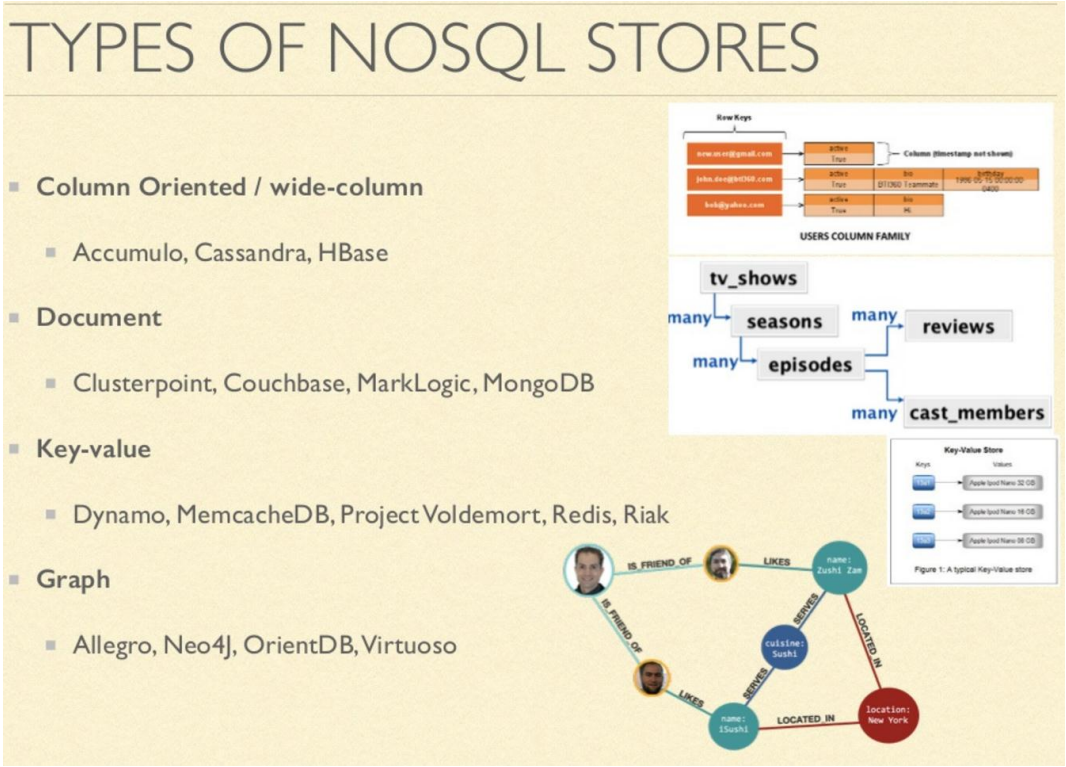


Ilustración 22 - Clasificación de Bases de Datos NoSQL

Las bases de datos NOSQL se pueden clasificar según su forma de almacenar los datos de la siguiente forma [29]:

- **Orientadas a columnas** → Este tipo de bases de datos están pensadas para realizar consultas y agregaciones sobre **grandes cantidades de datos**. Funcionan de forma parecida a las bases de datos relacionales, pero almacenando columnas de datos en lugar de registros. Ej: **Cassandra, HBase**…

- **Documentales** → Son aquellas que gestionan **datos semi-estructurados**, es decir, documentos. Estos datos son almacenados en algún formato estándar como puede ser **XML, JSON o BSON**. Son las bases de datos NOSQL **más versátiles**. Se pueden utilizar en gran cantidad de proyectos, incluyendo muchos que tradicionalmente funcionarían sobre bases de datos relacionales. Ej: MongoDB, CouchDB, **Firebase, Google Datastore** …

- **Clave-Valor** → Estas son **las más sencillas de entender**. Simplemente **guardan tuplas que contienen una clave y su valor**. Cuándo se quiere recuperar un dato, simplemente se busca por su clave y se recupera el valor. Ej: DynamoDB, Redis, **Riak, MemcacheDB**…

- **Orientadas a grafos** → Basadas en la **teoría de grafos, utilizan nodos y aristas para representar los datos almacenados.** Son muy útiles para guardar información en modelos con muchas relaciones, como redes y conexiones sociales. Ej: **Neo4j, Infinite Graph,** …

3.6.5. BBDD DOCUMENTALES

Ahora vamos a hablar un poco sobre **BBDD documentales** debido a que son las que se han tenido más en consideración para utilizar en el proyecto, y que al final ha sido una BD de este tipo la que se ha utilizado (Firebase) [33].

Las **BBDD documentales**, como hemos visto en el punto anterior, almacenan **datos semi-estructurados** conocidos como **documentos**. El **formato** para almacenar estos datos suele ser **XML, JSON o BSON** principalmente, aunque también se pueden utilizar **YAML, formatos binarios** como PDF o documentos de MS OFFICE. **En este proyecto** la BD que hemos utilizado (**Firebase**) **utiliza JSON**.

Un documento en una BD documental **es una agrupación de información**, es decir, la codificación de una información. Esto ha supuesto una gran ventaja, ya que nos permite almacenar un objeto de nuestra aplicación directamente en la BD y que al recuperar un documento, se pueda transformar en un objeto de nuestro sistema. Todo esto gracias a la API que nos proporciona Firebase para transformar clases en objetos JSON que después pueden ser almacenados en la BD.

Una **ventaja** que tienen los **documentos** es que **no tiene por qué seguir la misma estructura** como pasaría por ejemplo en una tupla de un RDBMS. Esto **nos permite tener documentos con una información que otros no tienen**. Por ejemplo podríamos tener un objeto que represente a una persona que no tenga hijos pero sí profesión y otro objeto que representa a otra persona que sí tiene hijos, pero no tiene profesión.

```
{
   Nombre:"Pepe",
   Dirección:"Plaza Mayor 5",
   Profesión:"Panadero"
}
```

```
{
    Nombre:"Juliana",
    Dirección:"Gran Vía 15",
    Hijos:[
       {Nombre:"Miguel", Edad:10},
       {Nombre:"Jacinta", Edad:8},
       {Nombre:"Sara", Edad:5},
       {Nombre:"Elena", Edad:2}
  ]
 }
```

Todo documento está asociado a una clave única que permite acceder a él de forma directa. **En Firebase la clave es una URI,** en otros sistemas la clave puede ser una simple cadena.

En cuanto a la recuperación de la información, las BBDD documentales suelen proporcionar una API que nos permite recuperar la información según ciertos parámetros, por ejemplo, recuperar todas las personas que en profesión tengan "Panadero".

Con respecto a la organización de los documentos en este tipo de BBDD, podemos encontrarnos con los siguientes **métodos de organización**:

- Colecciones
- Etiquetas
- Metadatos ocultos
- Jerarquías de directorios

3.6.6. APORTE AL PROYECTO

La BBDD elegida, en este caso **Firebase**, juega un papel fundamental en la aplicación, ya que como se contará con más detalle en el apartado Elección de BD, esta BD aporta muchos beneficios a la aplicación como por ejemplo, modo offline, cache, autenticación, API RESTful, … Es verdad que esta BD no fue la primera en ser elegida, y en el apartado comentado anteriormente podemos ver cuál fue la primera alternativa y por qué fue descartada.

La elección de una BD de tipo NOSQL viene principalmente porque, como se ha expuesto en este apartado, este tipo de bases de datos son las más apropiadas para uso futuro en aplicaciones móviles en las cuales se van a manejar grandes cantidades de información. **Pensando en un futuro, esta es la mejor elección gracias a las grandes posibilidades que ofrecen estas BBDD para escalar horizontalmente**. El **punto negativo** de estas BBDD está en la **consistencia de los datos**, y para ello hay que tener estos datos bien organizados.

3.7. DISEÑO DE MATERIALES (MATERIAL DESIGN)

3.7.1. ¿QUÉ ES MATERIAL DESIGN?

Material design es una normativa de diseño enfocado en la visualización del sistema operativo Android, además en la web y en cualquier plataforma. Fue desarrollado por Google y anunciado en la conferencia Google I/O celebrada el 25 de junio de 2014.

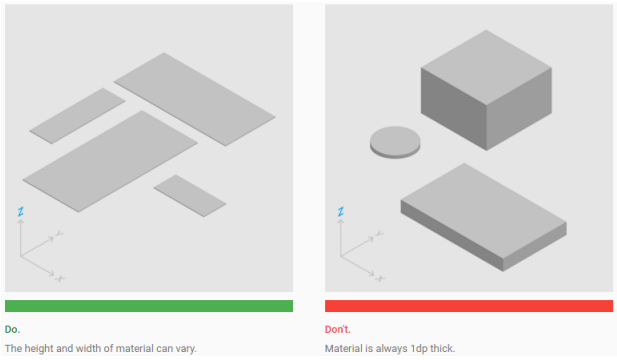
La filosofía de esta nueva normativa de diseño es la de diseñar aplicaciones simulando que la aplicación está compuesta por materiales. Estos materiales tienen una serie de propiedades como por ejemplo la profundidad (altura) y que el grosor del material siempre es de 1dp. Este nuevo diseño propuesto por Google busca unificar el diseño de las aplicaciones con el objetivo de dar una mejor experiencia al usuario.

Antes de que Google anunciase Material design, no se seguían unas normas bien definidas, había un diseño llamado Holo que definía cómo se hacían ciertas partes de la aplicación, como por ejemplo la toolbar, pero no definía cómo se debería mostrar el contenido de la aplicación, o cómo debería reaccionar la aplicación ante diferentes acciones del usuario. Esto en material design está mejor definido y define una guía de estilo completa de cómo se deben diseñar las aplicaciones para conseguir una mejor experiencia de usuario. Todo esto podemos verlo en la guía de Material Design que Google pone a nuestra disposición [34].

3.7.2. CARACTERÍSTICAS PRINCIPALES DEL MATERIAL

A continuación se va a mostrar una pequeña lista con las **principales características y comportamientos** de este diseño basado en materiales:

- El material puede variar en alto y ancho, pero no en **espesor**, que **siempre será de 1dp**.

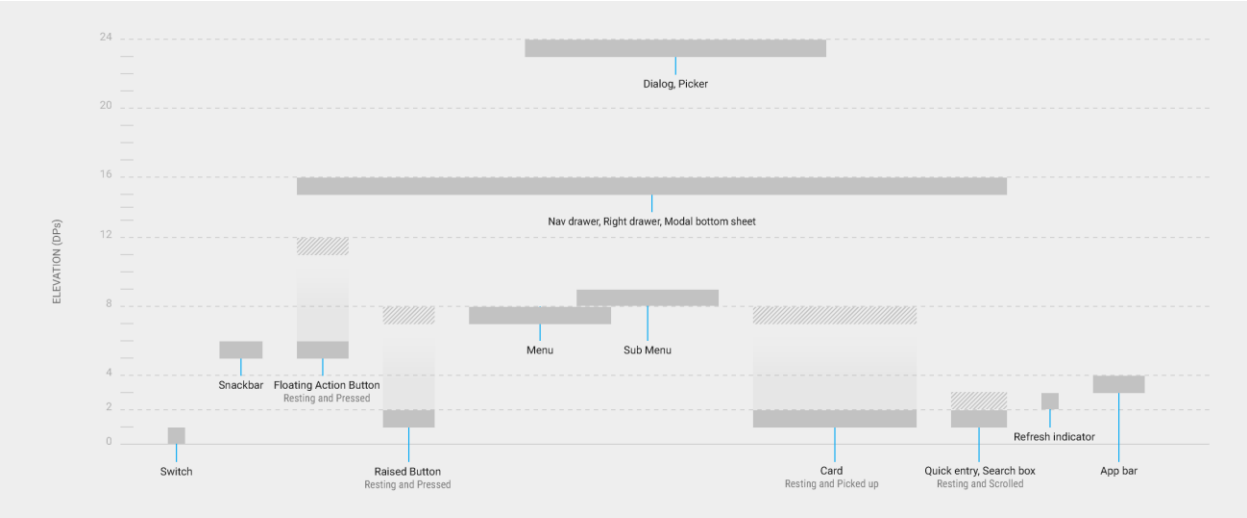


- El contenido se pinta sobre el material sin añadir espesor.
- El contenido puede comportarse de forma diferente al material, pero está limitado a los límites del material.
- El material es sólido, los eventos no pueden pasar entre materiales.

- No pueden existir varios materiales ocupando el mismo espacio simultáneamente.

- El material puede cambiar de forma
- El material crece y se reduce solo a lo largo de su plano.
- **El material nunca se dobla o pliega**
- El material puede unirse a otros materiales y convertirse en una única pieza de material
- El material puede dividirse
- El material puede moverse a lo largo de cualquier eje
- El movimiento a través del eje Z es la respuesta típica del material a una interacción por parte del usuario con el material

Material design también **define** cuáles deben ser **las elevaciones** que debe tener cada elemento de la aplicación **según el tipo de elemento**. En la siguiente imagen se recogen los principales elementos que forman parte de una aplicación con su correspondiente elevación.



3.7.3. COLORES

En material design se ha tenido muy en cuenta la **teoría del color** para que la aplicación tenga un aspecto agradable a la vez que se diferencian bien las distintas partes de ésta [35].

En **material design** una aplicación tiene que tener **al menos dos colores básicos** que son el **color primario** de la aplicación y el **color de acentuación**. Estos dos colores deben ser distintos lógicamente pero no deberían ser cualquier combinación de colores, aquí es donde entra en juego la **teoría del color** o qué colores son los que mejor combinan entre sí. Para esto hay muchas webs que te ayudan en la elección de estos colores, **Material Palette** es una de ellas [36].

En esta web nosotros podemos seleccionar dos colores (color primario y color de acentuación) y él nos generara automáticamente una previsualización de cómo quedaría una aplicación con esos colores y nos generará una serie de archivos que nos permitirá descargar nuestra paleta de colores. Esta paleta de colores se compone de varios colores que Android podrá utilizar para distintos elementos y para dar un mejor aspecto a nuestra aplicación. Por ejemplo:

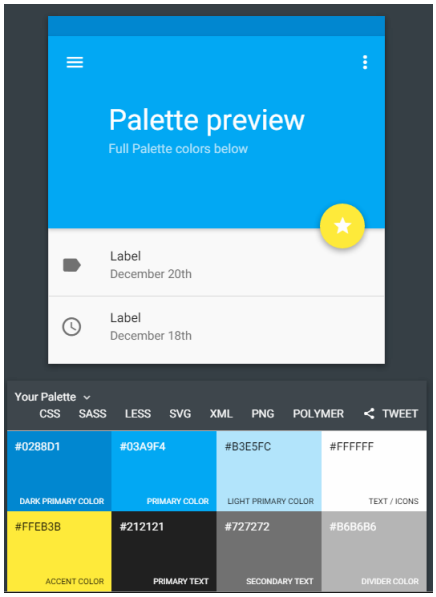


Ilustración 23 - Ejemplo de Material Palette

3.7.4. ¿POR QUÉ MATERIAL DESIGN?

Realizar una aplicación siguiendo estas normas de diseño ha sido principalmente porque como se ha comentado antes, seguir esta guía de estilo nos va a permitir que el usuario se encuentre más satisfecho y cómodo utilizando la aplicación, que le resulte más atractiva y que sepa interactuar con ella, reaccionando correctamente a los estímulos del usuario.

Cada vez es mayor la acogida que está teniendo material design entre los desarrolladores de aplicaciones móviles y este será el diseño que se va a utilizar en los próximos años en Android. Por esto, seguir esta guía de diseño nos asegura que en un futuro no tendremos que rediseñar toda la aplicación, ya que tendremos las bases para construir un mejor diseño sobre el actual. También, que debido a que esta aplicación está basada en API mínimo 21 (Lollipop), y que fue en Lollipop donde se introdujo material design en Android, esto va a permitir que el usuario no se note extraño al utilizar la aplicación, ya que seguirá las mismas normas de estilo que se siguen para diseñar Android.

3.7.5. APORTE AL PROYECTO

El aporte al proyecto de esta guía de estilo es bastante grande, ya que nos da las normas para diseñar una aplicación que va a estar en armonía con el resto del sistema operativo y el resto de aplicaciones y va a ser muy fácilmente entendible por los usuarios.

A parte de la interacción con el usuario, a mí personalmente me parece que los nuevos elementos que proporciona material design son muy amigables y que el resultado final de una aplicación diseñada siguiendo estas normas que nos marca Google es genial.

En definitiva, utilizar material design ha hecho que mi aplicación sea más usable y tenga un mejor aspecto.

Más adelante en el documento se mostraran los principales Elementos de Material Design utilizados en el proyecto.

4. TOMA DE DECISIONES

En este apartado se va a dejar reflejado una serie de decisiones que se han ido tomando a lo largo del desarrollo del proyecto, principalmente sobre las tecnologías utilizadas.

4.1. ELECCIÓN DE BEACONS Y LIBRERÍA

4.1.1. ELECCIÓN DEL BEACON

Una de las primeras cosas que se hicieron al empezar este proyecto fue investigar qué era un Beacon y qué opciones había en el mercado. Cuando se empezó esta investigación, en el mercado no había muchas empresas reconocidas que vendieran este tipo de dispositivos, pero se dio con unas cuantas sobre las que se podía elegir.

Una de las primeras empresas que se tuvo en cuenta por proximidad fue **Remotte**, una pequeña Startup extremeña que se ha encargado de utilizar este tipo de dispositivos para modernizar la zona antigua de Cáceres. Esta empresa al final no fue la elegida por una cuestión de precio, ya que era aproximadamente el doble que el del Beacon elegido, y porque el dispositivo elegido es también compatible con **Physical Web**, que es la nueva forma que propone **Google** de utilizar estos dispositivos para proporcionar información basada en el contexto y que a diferencia de los **Beacons**, en **Physical Web** lo que transmite el dispositivo es una **URL**.

Otra empresa que se tuvo en cuenta fue **Estimote**. Esta empresa tiene mucho prestigio en la construcción de este tipo de dispositivos, ya que las personas que los han usado dan muy buenos comentarios de ellos. Otra cosa buena que tienen estos **Beacons**, es que la empresa proporciona una **API** para facilitar el acceso a las características que ofrecen estos dispositivos. El **punto negativo** por el que no fue esta la empresa elegida para comprar el **Beacon**, fue que es una empresa de **EEUU**, con lo cual el precio era bastante superior y se encontró otra empresa española que también tenía muy buenos comentarios y salían bastante más económicos.

La última empresa consultada y que fue donde finalmente se compró los Beacons fue **Accent Systems [39]**. Esta es una empresa ubicada en **Barcelona** que fabrica todo tipo de dispositivos electrónicos, entre ellos **Beacons**. Esta empresa nos **permite personalizar varios aspectos del Beacon**, por ejemplo nos permite elegir un **logo**, nos permite personalizar el **formato** del Beacon, el **hardware,** … El **Beacon** elegido para el proyecto es el **iBKS 105** aunque podría haber sido cualquier otro de los que ofrece esta empresa, pero este es compatible con **Physical Web**, que aunque no haya sido utilizado en este proyecto, podría utilizarse en un futuro. El precio que ofrece esta empresa era también bastante barato con respecto a las anteriores.

	Remotte	Estimote	Accent Systems
Ubicación	**Cáceres**	**EEUU**	**Barcelona**
Precio	≈ 30 €/u	≈ 20 €/u (Pack de 3) + Gastos de Envío	≈ 20 €/u (Pack de 3)
Características	iBKS 105 Beacon + Physical Web	iBeacon + Api	iBKS 105 Beacon + Physical Web
Comentarios	Son los mismos que vende Accent Systems	Muy buenos resultados y fáciles de utilizar	Muy buenos resultados (Utilizados en la Universidad)
Decisión Final	✗	✗	✓

Tabla 3 - Elección Beacon

4.1.2. ELECCIÓN DE LA LIBRERÍA

Una vez elegido el Beacon decidí buscar una librería que me facilitase la **detección del Beacon** (conocido como **Monitoring**) y la **determinación de la distancia** a la que se encuentra el Beacon (conocido como **Ranging**).

Buscando un poco en internet encontré las siguientes opciones: Estimote Beacons API y Android Beacon Library [37, 38]. De estas opciones la que más me gustó fue **Android Beacon Library** por ser una librería de código abierto y ser compatible con Beacons de distintos fabricantes (mediante una clase llamada **BeaconParser**). A parte de esto, es la librería de referencia para trabajar con **Beacons** y, como se puede ver en su web, esta librería está siendo utilizada por varias empresas en sus proyectos de Beacons.

	Android Beacon Library	Estimote Beacons API
Open Source	✓	✗
Compatibilidad con la mayoría de Beacons	✓	?
Recomendada por la Comunidad Android	✓	✗
Decisión Final	✓	✗

Tabla 4 - Elección de Librería Beacon

4.2. ELECCIÓN DE BD

Para la elección de la BD existían una serie de restricciones que ésta debería cumplir, como que fuese una **BD en la nube** para que se pudiese **acceder a la información desde distintos dispositivos** simultáneamente si fuese necesario, que fuese **fácil de utilizar** y que tuviese **versión gratuita** para poder desarrollar el proyecto.

Esta elección fue bastante importante y supuso un gran retraso en el proyecto no haberla elegido bien desde el principio. A continuación se explicará cuál era la **primera opción** que se intentó utilizar pero al final no fue posible, y que ha supuesto un gran retraso en este proyecto, ya que se dedicó bastante tiempo a trabajar con esta solución hasta que se comprobó que no se podía continuar. Cuando se percibió que la solución anterior no era posible, se decidió buscar otra solución y esta sí ha sido la que se ha utilizado finalmente.

4.2.1. APPENGINE + OBJECTIFY + RETROFIT. LA OPCIÓN QUE NO PUDO SER.

Como primera opción para almacenar los datos para este proyecto, se pensó en utilizar la **arquitectura Cloud** que ofrece **Google**, concretamente el **DataStore** que se encuentra dentro del paquete de **AppEngine** y que es gratuito hasta cierto volumen de tráfico... [49]

CLOUD DATASTORE PRICING

Cloud Datastore is a highly-scalable NoSQL database for your web and mobile applications

	FREE LIMIT PER DAY	PRICE ABOVE FREE LIMIT
Stored data	1 GB storage	$0.18 / GB / month
Read Operations	50k operations	$0.06/100k operations
Write Operations	50k operations	$0.06/100k operations
Small Operations	Unlimited	Free

Tabla 5 - Cloud DataStore Pricing

En un principio, esta opción cumplía los requisitos comentados anteriormente (DB en la nube, fácil y con versión gratuita), por lo que se comenzó a investigar cómo funcionaba el almacenamiento y la extracción de esta BD.

Consultando diversas webs, se pudo ver que el **DataStore** de **Google** ofrece 3 formas de acceso que son **Java Data Objects (JDO), Java Persistence API (JPA) y una versión de más bajo nivel** desarrollada por **Google** (aunque nadie recomienda esta última).

Debido a que parecía un poco complejo en ese momento aprender estas tecnologías, se buscó si existía alguna especie de **ORM** que facilitase un poco el acceso y almacenamiento en esta BD y se encontró **Objectify**, que es una librería que en base a anotaciones permite definir cómo se mapean los objetos en la BD y tiene una serie de operaciones que permite realizar las operaciones **CRUD** en la BD de forma sencilla [45]. De hecho, consultando la documentación oficial de **Google**[19], se puede ver como recomienda el uso de **Objectify** frente a **JPA o JDO** (cuando se comenzó este proyecto no estaba recomendado de forma oficial por **Google**):

App Engine Datastore

App Engine Datastore is a schemaless object datastore providing robust, scalable storage for your application, with the following features:

- Highly reliable and covered by the App Engine SLA.
- ACID transactions.
- Advanced querying features.
- High availability of reads and writes.
- Strong consistency for reads and ancestor queries.
- Eventual consistency for all other queries.

You can access Datastore using the low-level API described throughout the Datastore documentation, which provides direct access to all of Datastore's features, or you can use one of the higher-level open-source APIs for Datastore that provide ORM-like features and a more abstract experience, such as Objectify.

Read more about App Engine Datastore on the App Engine Datastore API documentation page.

Tabla 6 - Recomendación de Objectify por Google

Una cosa a tener en cuenta es que **AppEngine es un servicio PaaS** y, como tal, necesita código de servidor para poder acceder a la BD. Para ello, **Google** lo hace un poco más fácil gracias a la integración de su API para acceder a **AppEngine** desde **Android**, ya que genera automáticamente los **EndPoint** necesarios para las entidades que definas mediante las anotaciones de **Objectify** con las operaciones **CRUD** básicas y, además, el código generado utiliza **Objectify** para el acceso al **DataStore**, eliminando la complejidad de utilizar **JPA** o **JDO**.

[19] Fuente: https://cloud.google.com/appengine/docs/java/storage

Objectify también permite asegurar un mínimo de consistencia gracias a las relaciones entre entidades. Esto permite definir una entidad que tiene una dependencia de otra para que se almacenen juntas, o que solo se almacene la clave de esa entidad para poder recuperarla al cargar la entidad o al acceder a ese dato. Esto fue muy interesante y una de los puntos que más motivó el uso de esta librería, ya que cuando se intentó utilizar, no estaba directamente recomendada por **Google** como lo está ahora.

Objectify tiene varias posibilidades para crear relaciones [46]. Básicamente permite dos tipos: **Key y Ref**. **Key** almacena solamente una clave de otra entidad, mientras que **Ref** hace referencia a otra entidad que será cargada cuando se acceda a ella o cuando se cargue el objeto si se utiliza junto con la anotación **@load**.

Con **AppEngine + Objectify** se disponía de la parte para **almacenar los datos y generar la api REST del servicio**, pero se necesitaba poder **consultar estos datos [40, 41, 42, 43]**. Para esta función se eligió una de las librerías más conocidas para esta labor: **RetroFit [44]**. Esta librería facilita mucho las labores de consulta a una **API REST** desde cualquier aplicación Java.

El **funcionamiento de RetroFit** es muy sencillo y un ejemplo sencillo consistiría en **3 sencillos pasos**:

1. Crear el **generador de servicios**. Esto es una pequeña clase donde definiremos la URL de nuestra **API** y que se encargará de construir una clase a la que podremos realizar las consultas como si se tratase de un objeto.

```
public class ServiceGenerator {

    public static final  String API_BASE_URL = "http://your.api-base.url";

    private static RestAdapter.Builder builder = new RestAdapter.Builder()
                .setEndpoint(API_BASE_URL)
                .setClient(new OkClient(new OkHttpClient()));

    public static <S> S createService(Class<S> serviceClass) {
        RestAdapter adapter = builder.build();
        return adapter.create(serviceClass);
    }
}
```

Tabla 7 - Generador de servicios RetroFit

2. Se crearán las **interfaces** que sean necesarias, donde se definirán las consultas necesarias para utilizar la **API REST**. Estas interfaces son las que se le pasarán al **generador de servicios** anterior para que generé los objetos, que permitirá consultar a la **API**, en base a las anotaciones de **RetroFit**. La siguiente imagen muestra un ejemplo de la definición de una interfaz para obtener los contribuidores de un repositorio de GitHub.

```
public interface GitHubClient {
    @GET("/repos/{owner}/{repo}/contributors")
    List<Contributor> contributors(
        @Path("owner") String owner,
        @Path("repo") String repo
    );
}
```

Tabla 8 - Ejemplo definición interfaz RetroFit

3. Por último, para utilizar la **API**, es necesario primero obtener el **cliente** definido previamente mediante el **generador de servicios,** y realizar las llamadas a los métodos definidos en el cliente. En la siguiente imagen se muestra un ejemplo:

```
public static void main(String... args) {
    // Create a very simple REST adapter which points the GitHub API endpoint.
    GitHubClient client = ServiceGenerator.createService(GitHubClient.class);

    // Fetch and print a list of the contributors to this library.
    List<Contributor> contributors =
        client.contributors("fs_opensource", "android-boilerplate");

    for (Contributor contributor : contributors) {
        System.out.println(
                contributor.login + " (" + contributor.contributions + ")");
    }
}
```

Tabla 9 - Obtención del cliente RetroFit

Como se puede ver, con tres sencillos pasos tenemos un ejemplo sencillo de consulta a una API REST. Ese es el gran potencial de esta API, la simplicidad. Esta API también permite realizar consultas a APIs con distintos métodos de autenticación (usuario/contraseña, OAuth o token), permite realizar llamadas de forma síncrona/asíncrona y mucho más.

Volviendo al tema de **Objectify**, el problema que existía cuando se estuvo utilizando esta API (Febrero de 2015) es que no funcionaban correctamente ni las claves ni las referencias, y esto suponía un gran impedimento a la hora de seguir utilizándola en el proyecto. A continuación pueden verse un par de enlaces donde aparecen estos problemas, que parece que ya están solucionados, pero en su momento no funcionaban las soluciones encontradas en internet:

- Primer problema con anotación **Key**: https://stackoverflow.com/questions/19707359/objectify-with-cloud-endpoints [Disponible a 30 de diciembre de 2015]

- Problema utilizando la anotación **Ref** en las entidades: https://stackoverflow.com/questions/24590518/gaeobjectify-parameterized-type-com-googlecode-objectify-ref-not-supported [Disponible a 30 de diciembre de 2015]

Los problemas comentados anteriormente, junto a que las posibles soluciones sugeridas en la red pasaban por reemplazar el sistema anterior por **Firebase** y consultándolo parecía que podría cumplir con las necesidades del proyecto, hicieron que se cambiase la forma de almacenar los datos en la aplicación. **Objectify,** aunque parecía tener futuro y mucho potencial, en ese momento tenía algunos problemas y **Firebase** parecía ser mucho más sencillo de utilizar, ya que es una solución **SaaS** y, por tanto, no hay que implementar/mantener un backend, entre otras ventajas que se comentan en el siguiente punto.

4.2.2. FIREBASE. EL GRAN VENCEDOR

Firebase, como dice el título de este punto, ha sido el gran vencedor en este apartado, ya que gracias principalmente a que es una **solución SaaS**, la facilidad de utilizarlo es infinitamente superior a la propuesta anterior con **AppEngine**. **Firebase** nos proporciona una **API para cada plataforma** en la que queramos desarrollar (**Multiplataforma**) y es muy sencilla de utilizar.

Firebase ofrece las siguientes ventajas:

- **Multiplataforma** → Tiene una **API** para los principales sistemas **(Web, IOS, Android, …)** además de ofrecer una **API RESTful**

- **BD en tiempo real** → **Firebase** se define como una **BD en tiempo real**, ya que permite, mediante su **API**, establecer callbacks que serán ejecutados cuando se produzca un cambio en un punto de la BD sobre la que se subscriba el cliente (Patrón Observer).

- **Caché** → Esta es una de sus mejores características, ya que **Firebase** ofrece un sistema de **caché automático** lo que permite reducir bastante los problemas que surgen al implementar un sistema de cachés (como por ejemplo el de la **sincronización**). Además, **Firebase permite configurar el tamaño reservado para este sistema de caché** (por defecto 10MB), y también permite definir ciertos datos que se mantendrán siempre en la cache para mejorar el acceso a estos datos y tenerlos siempre disponibles (esto está relacionado con la siguiente funcionalidad).

- **Modo offline** → **Firebase** ofrece un **modo offline automático** utilizando para ello una **BD local y la BD en la nube**. Gracias a la funcionalidad anterior (caché), se puede acceder a datos consultados previamente si no tenemos conexión, y permite realizar cambios en la **BD local** que serán sincronizados con la **BD en la nube** cuando el usuario tenga conexión.

- **Autenticación** → **Firebase** ofrece una forma muy sencilla de añadir un **sistema de autenticación** en nuestra aplicación. Permite **varios tipos de autenticación**: Login/Password, Google, Facebook, Twitter, Github, Custom (Generar tokens personalizados) y anónimo (basado en sesiones).

- **Seguridad** → Con respecto a la seguridad, **Firebase** ofrece una configuración basada en un fichero **JSON** en el que se definen una serie de reglas/restricciones que se aplicarán al intentar acceder a los datos.

- **Panel de administración** → **Firebase** nos ofrece una forma muy sencilla de poder administrar todas estas funcionalidades en un **panel de administración** en el que tendremos acceso a la **información almacenada en la BD, a los usuarios logueados y a la configuración de seguridad** entre otras cosas.

Como ya se ha comentado en este documento, **Firebase** es una **BD NoSQL orientada a documentos**, concretamente almacena ficheros **JSON**. Un problema común en este tipo de BBDD es la **consistencia**, ya que este tipo de BBDD está pensada para almacenar datos **denormalizados**. Para poder tener relaciones entre los datos y asegurar así un poco la consistencia y evitar las duplicidades, **Firebase** recomienda utilizar unas ciertas normas para organizar los datos. Firebase tiene un artículo donde hablan de la **denormalización** de los datos y cómo hay que controlar las operaciones para mantener la **consistencia** de la BD [47].

Otra característica importante de **Firebase** es que tiene un plan gratuito que me ha permitido poder probarlo y desarrollar el proyecto utilizando esta BD. A continuación se muestra el **plan de precios** de **Firebase [48]**:

	Free	Spark	Candle	Bonfire	Blaze	Inferno
	$0 forever	$5 per month	$49 per month	$149 per month	$449 per month	$1,499 per month
REALTIME DATABASE						
Connections	100	100	UNLIMITED*	UNLIMITED*	UNLIMITED*	UNLIMITED*
Storage	1 GB	1 GB	10 GB	30 GB	100 GB	300 GB
Transfer	10 GB	10 GB	50 GB	150 GB	500 GB	1.5 TB
Private Backups	✗	✗	✗	✓	✓	✓
AUTHENTICATION						
Users	UNLIMITED	UNLIMITED	UNLIMITED	UNLIMITED	UNLIMITED	UNLIMITED
HOSTING						
Storage	1 GB	1 GB	10 GB	10 GB	10 GB	10 GB
Transfer	100 GB	100 GB	1 TB	1 TB	1 TB	1 TB
Custom Domain	✗	✓	✓	✓	✓	✓
	Sign Up	Purchase	Purchase	Purchase	Purchase	Purchase

Tabla 10 – Plan de precios de Firebase

Como se puede ver en la imagen anterior, la versión gratuita ofrece unas características que permitirían incluso probar este proyecto en un pequeño entorno real, ya que permite hasta **100 conexiones simultaneas**, con un **almacenamiento de 1 GB**, una **transferencia total de 10 GB** (en caso de **base de datos en tiempo real**) o **100 GB en caso de solo utilizarlo como hosting** (en este proyecto sería una base de datos en tiempo real).

Como conclusión final, en la siguiente tabla se pueden consultar las diferencias entre los sistemas contemplados para el almacenamiento de los datos para este proyecto.

	Firebase	Objectify + Google App Engine
Tipo de solución Cloud	**SaaS**	**PaaS**
Sin Backend	✓	✗
API para Clientes	✓	✗
API Multiplataforma	✓	✗
RESTful	✓	✓
BD en tiempo real	✓	✓
Caché	✓	✗
Modo offline	✓	✗
Varios sistemas de Autenticación	✓	✗
Securizacion de acceso a los datos	✓	✗
Panel de administración	✓	✓
Decisión Final	✓	✗

Tabla 11 - Firebase vs Objectify + Google App Engine

4.3. UNA APLICACIÓN O DOS

Este fue otro debate que hubo que solucionar, si se desarrollaban dos aplicaciones, una para los clientes y otra para los propietarios de los establecimientos. Por una parte, definir dos aplicaciones tenía la parte buena de que podría tener bien separado las necesidades de cada tipo de usuario y ser más específico en las funcionalidades que iban a necesitar, pero la parte negativa es que habría que llevar el mantenimiento de dos aplicaciones distintas donde habría muchas partes en común, y esto haría más difícil el mantenimiento en un futuro.

Al final, después de revisar las partes positivas y negativas de cada opción, se decidió que era mejor realizar una sola aplicación en la que distinguiría el tipo de usuario logueado apareciendo distintas opciones según el usuario. Esto puede parecer más complicado, pero al final hay mucha parte del código que se puede reutilizar en ambas interfaces y esta opción permitirá que en un futuro solo haya que mantener una aplicación.

4.4. TRANSFERENCIA DEL ID DEL USUARIO (NFC Y QR)

La forma de **transferir el identificador de usuario** a la hora de registrar una compra también fue un asunto que hubo que pensar para dar con la forma que sería más fácil para el usuario y el propietario del establecimiento.

Teniendo en cuenta que lo que se está desarrollando es una aplicación móvil, y que cada vez es más común que los móviles incluyan tecnología **NFC**, parece que esta sería la mejor forma de transferir el identificador de usuario.

Pero una vez decidido que **NFC** será la tecnología elegida para la identificación del usuario, surgieron otras preguntas, ¿qué pasa con los usuarios que no tengan NFC en sus móviles? ¿Y con los usuarios que se les olvide el dispositivo móvil? Para solucionar estas preguntas, surgió la idea de utilizar como **alternativa a NFC los códigos QR**, que ya son bastante conocidos y cualquier dispositivo móvil que tenga una cámara podría leerlos.

Además, los **códigos QR** también solucionan el problema de que una persona quiera registrar una compra a su nombre pero se le haya olvidado el móvil en casa o directamente no tenga dispositivo móvil y quiera acumular puntos para poder conseguir los premios que ofrezcan los establecimientos por estos puntos. La idea es que el usuario pueda generar en la aplicación un **código QR** que le identifique como usuario dentro de la plataforma, y que también los establecimientos puedan entregar a los usuarios "offline" sus tarjetas con su código QR (previo registro en la plataforma).

Resumiendo, existen **dos alternativas para la identificación de un usuario** a la hora de asignarle una compra, el uso de la **tecnología NFC** o la lectura de un **código QR** que puede ser generado por la aplicación, o leído de una tarjeta de fidelización.

4.5. USO DE UN REPOSITORIO GIT

Esto en realidad no fue una decisión exclusiva del proyecto, sino más bien una forma de tener el proyecto bien organizado y tener un sitio centralizado donde poder seguir la evolución del proyecto.

Como repositorio GIT se ha utilizado **BitBucket**, gracias a que ofrece una versión gratuita que permite almacenar hasta 5 proyectos GIT privados, permitiéndonos compartir el repositorio con otros usuarios (hasta un máximo de 5 usuarios), en este caso se ha compartido con José María Conejero, que ha sido el tutor de este proyecto.

5. MANUAL DEL PROGRAMADOR

En este apartado se explica con un poco más de detalle cómo se ha llevado a cabo el desarrollo del proyecto. Se tratan temas como los requisitos del sistema, arquitectura o diseño e implementación. En estos puntos se tratará la estructura del proyecto, cuáles son sus componentes principales y las librerías que se han utilizado. También se explicarán algunos **patrones de diseño** utilizados y qué beneficios aportan.

5.1. REQUISITOS DEL SISTEMA

Los requisitos del sistema serán los siguientes:

- La aplicación debe permitir el **registro de usuarios** y tendrá una opción para distinguir entre los clientes y los propietarios
- La aplicación debe permitir el **acceso tanto a clientes como a propietarios,** mostrándoles unas opciones u otras según el tipo de usuario
- La aplicación debe permitir a un cliente **registrar una compra** en un establecimiento **mediante NFC o Código QR,** y al propietario le tiene que permitir poder leer el código que genere el usuario o introducirlo manualmente para poder registrar la compra
- La aplicación debe permitir a un propietario crear nuevas promociones
- La aplicación debe permitir a un cliente consultar las promociones de las tiendas
- La aplicación debe mostrar una notificación al pasar cerca de un Beacon que tiene asociado un anuncio de una tienda registrada
- La aplicación debe permitir a los clientes visualizar los puntos que tienen en cada tienda y los puntos generales
- La aplicación debe permitir a los propietarios visualizar la información de los usuarios junto con sus puntos.

5.1.1. DIAGRAMAS DE CASOS DE USO

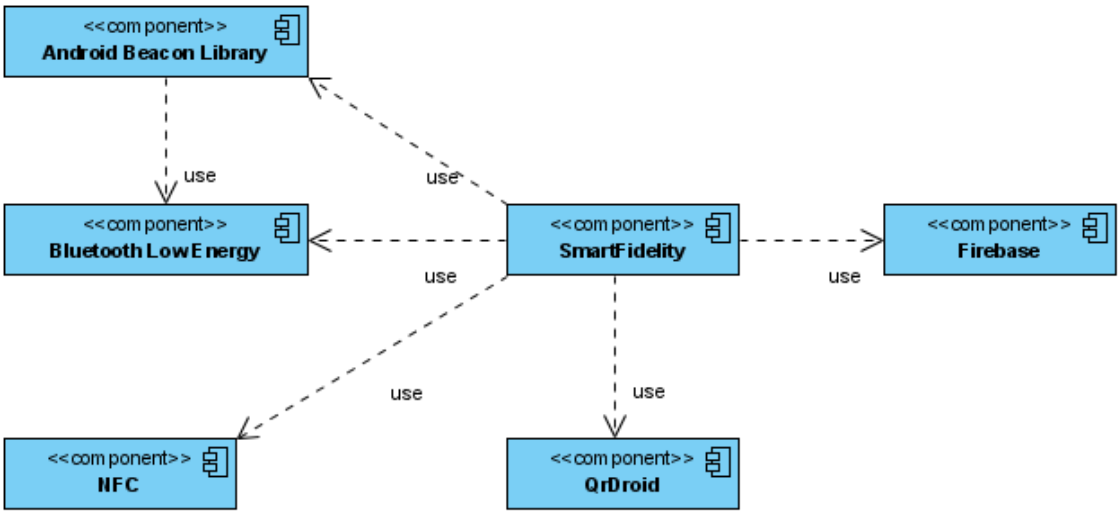
Ilustración 24 - Diagrama de Casos de uso

5.2. ARQUITECTURA

Ilustración 25 - Tecnologías implicadas

5.2.1. DIAGRAMA DE COMPONENTES

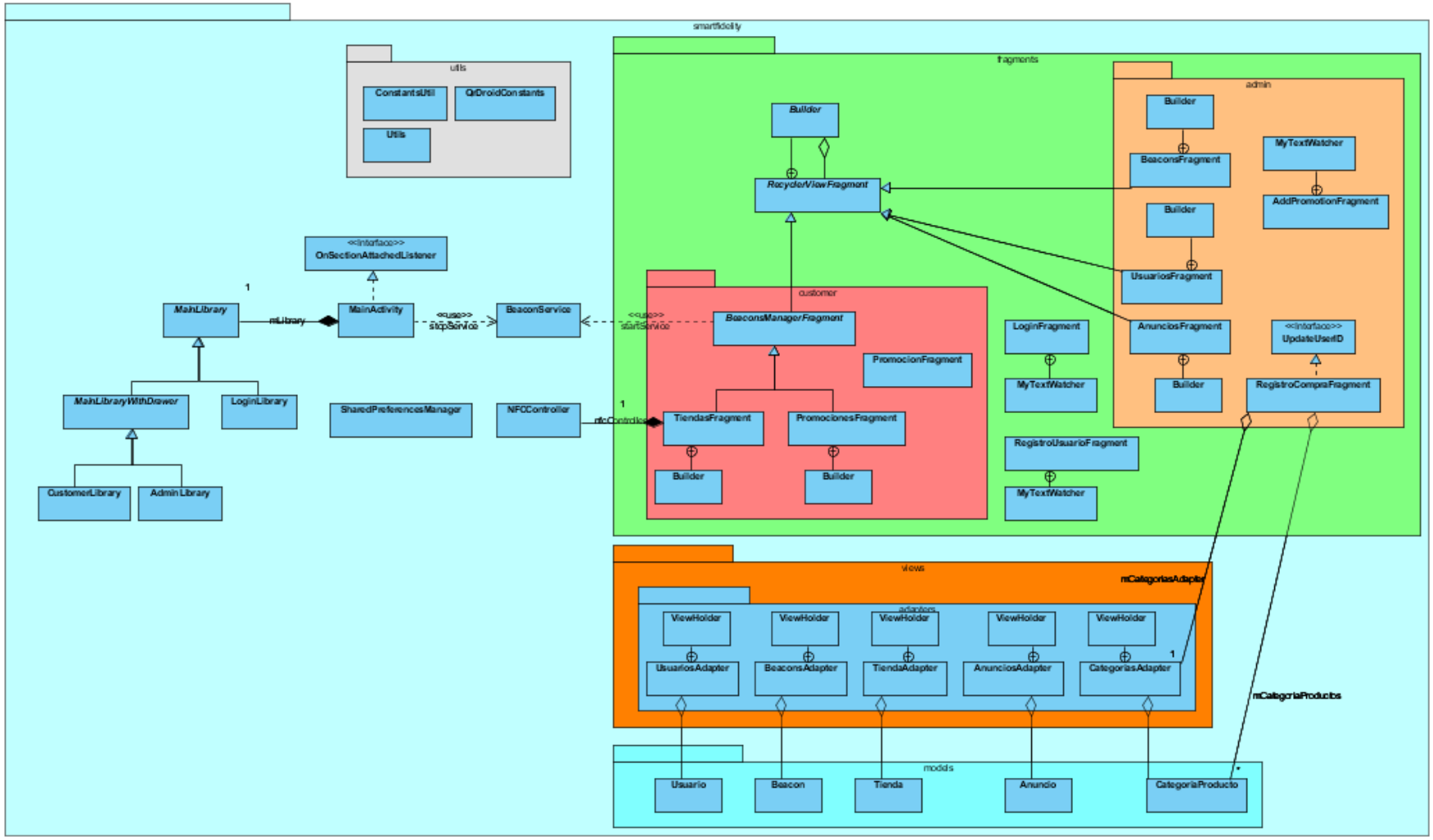
En la siguiente imagen se puede ver de forma general cuáles son los principales componentes que forman la aplicación. Este diagrama servirá para tener una visión más general de cómo está estructurada la solución.



Por un lado se encuentra en el centro la aplicación. Ésta tiene a su derecha **Firebase**, que no es solo una **Base de datos**, ya que también nos proporciona otras características como **Autenticación, Caché, Modo offline**... A su izquierda vemos el **Bluetooth Low Energy**, que se utiliza junto con la librería **Android Beacon Library** para la detección de **Beacons**. Por la parte inferior nos encontramos el **NFC** y **QrDroid** (aplicación que proporciona un servicio para generar códigos QR), utilizados para la **transferencia del identificador de usuario** entre aplicaciones.

5.2.2. ESTRUCTURA DEL PROYECTO

En este apartado se va a ir explicando por partes el **diagrama de clases** del proyecto, comentando las decisiones tomadas en cada momento. La siguiente imagen muestra una visión global del diagrama de clases que se explicará en este apartado.



MAIN ACTIVITY

En la siguiente imagen se puede ver como a la derecha se encuentra la **MainActivity** que es la actividad principal de la aplicación que carga las diversas pantallas según la situación que corresponda en cada momento (**Controlador**). Ésta tiene un atributo llamado **mLibrary,** este atributo es del tipo **MainLibrary** y se utiliza en el código para realizar la carga de la aplicación según el estado actual de la aplicación (**Patrón de diseño Strategy**). Si vemos la jerarquía de clases, en este momento tenemos tres clases que se pueden instanciar: **LoginLibrary, CustomerLibrary y AdminLibrary.** Como se puede ver en la imagen, **LoginLibrary** extiende de **MainLibrary** directamente, mientras que las otras dos extienden de **MainLibraryWithDrawer**, que es una clase abstracta que extiende la funcionalidad de **MainLibrary** añadiendo el **Navigation Drawer**, con lo que cualquier clase que extienda **MainLibraryWithDrawer** tendrá el **Navigation Drawer**, además de la **Toolbar** y el **Floating Action Button (FAB),** que también han sido añadidos a esta clase.

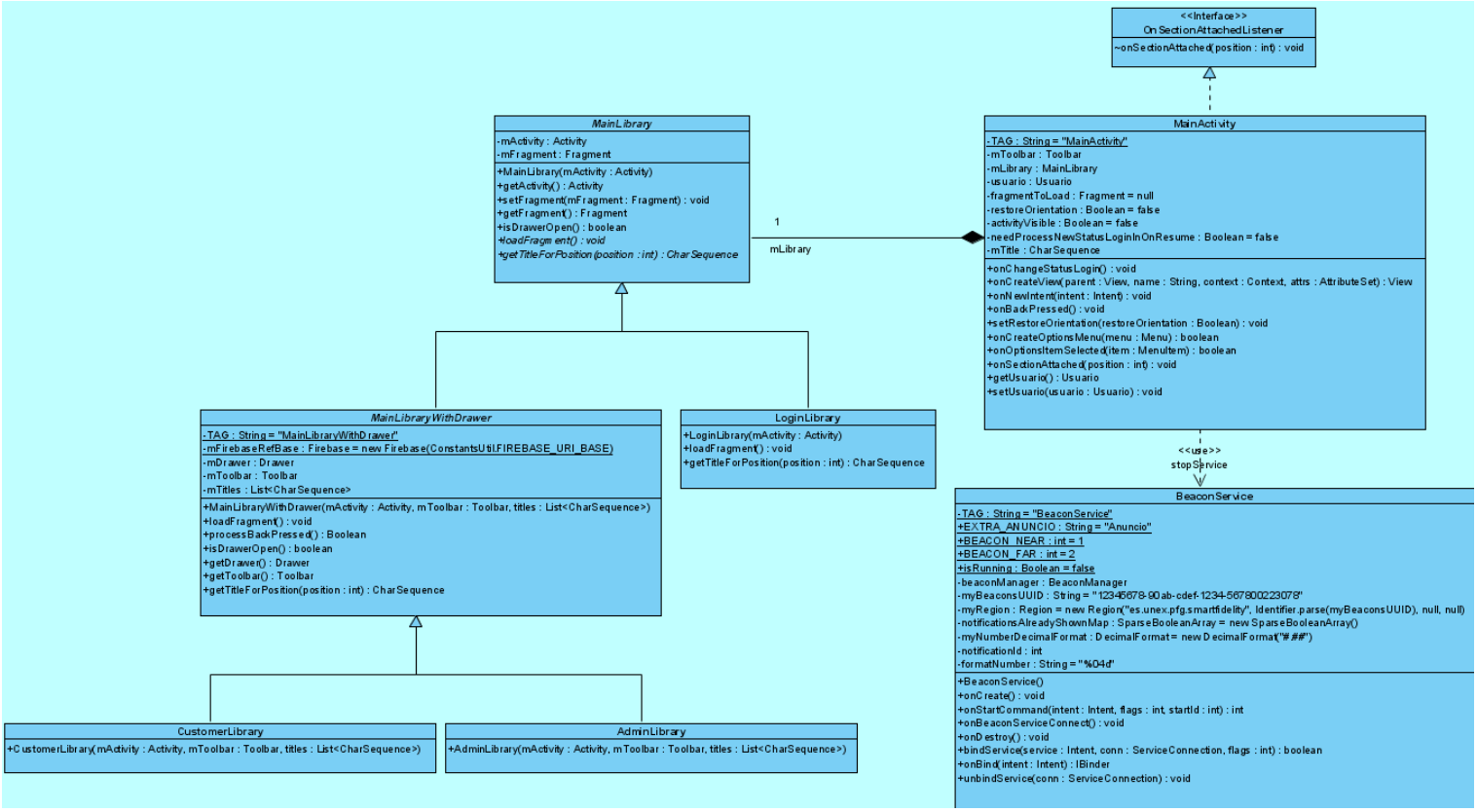


Ilustración 26 - MainActivity

En la siguiente imagen se puede ver la ventaja de haber utilizado este **patrón de diseño** (**Strategy**). Según el estado del login del usuario se instancia el atributo con una clase de la jerarquía u otra, y luego llamo al método **loadFragment** para cargar el fragmento que corresponda según esté definido en la clase instanciada. El siguiente código ha sido sacado de un método llamado **onChangeStatusLogin** que se encuentra en la **MainActivity** y que es llamado cada vez que se cambia el estado de login del usuario (se loguea/desloguea), cambiando dinámicamente el funcionamiento de **mLibrary** según corresponda en cada caso.

```
switch (statusLogin) {
    case ConstantsUtil.USER_NOT_LOGGED:
        Log.d(TAG, "Deshabilitando sensor de orientacion");
        setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_NOSENSOR);
        mLibrary = new LoginLibrary(this);
        break;
    case ConstantsUtil.CUSTOMER_USER_LOGGED:
        titles.add("Principal");
        titles.add("Promociones");

        mLibrary = new CustomerLibrary(this, mToolbar, titles);
        break;
    case ConstantsUtil.ADMIN_USER_LOGGED:
        titles.add("Registro");
        titles.add("Anuncios");
        titles.add("Beacons");
        titles.add("Usuarios");

        mLibrary = new AdminLibrary(this, mToolbar, titles);
        break;
    default:
        Log.d(TAG, "ERROR. This should not happen");
        mLibrary = new LoginLibrary(this);
}
mLibrary.loadFragment();
```

Ilustración 27 – Ejemplo de uso del patrón Strategy

En este punto, al llamar al **loadFragment** si la clase instanciada hereda de **MainLibraryWithDrawer,** deberá implementar un método llamado **getFragmentToLoad** que será el fragmento a cargar según la posición seleccionada en el **Navigation Drawer.** En la siguiente imagen podemos ver la implementación del método **loadFragment** en la clase **MainLibraryWithDrawer.**

```
@Override
public void loadFragment() {
    Fragment fragmentToLoad = getFragmentToLoad();
    if (fragmentToLoad != null) {
        setFragment(fragmentToLoad);
        getActivity().getFragmentManager().beginTransaction().replace(R.id.act_main_container, fragmentToLoad).commitAllowingStateLoss();
    }
}
```

Ilustración 28 - Ejemplo carga loadFragment en MainLibraryWithDrawer

Como se puede ver en la siguiente imagen, lo primero que se hace es obtener el fragmento a cargar del método **getFragmentToLoad**.

```
@Override
protected Fragment getFragmentToLoad() {
    Fragment fragmentToLoad = null;
    int position = getDrawer().getCurrentSelection();

    switch (position) {
        case 0:     // Registrar Compra y Tiendas (Principal)
            List<Tienda> mListTiendas = new ArrayList<~>();
            fragmentToLoad = new TiendasFragment.Builder()
                    .withSectionNumber(position)
                    .withList(mListTiendas)
                    .withAdapter(new TiendaAdapter(mListTiendas, SharedPreferencesManager.getUidUserLogged(getActivity()
                    .withLayoutFragment(R.layout.fragment_tiendas)
                    .withSwipeContainer(R.id.frag_tiendas_swipeContainer)
                    .withReciclerView(R.id.frag_tiendas_recyclerView)
                    .withEmptyList(R.id.frag_tiendas_empty_list)
                    .build();
            break;
        case 1:     // Anuncios (Promociones)
            List<Anuncio> mListAnuncios = new ArrayList<~>();
            fragmentToLoad = new PromocionesFragment.Builder()
                    .withSectionNumber(position)
                    .withList(mListAnuncios)
                    .withAdapter(new AnunciosAdapter(mListAnuncios))
                    .withLayoutFragment(R.layout.fragment_promociones)
                    .withSwipeContainer(R.id.frag_promociones_swipeContainer)
                    .withReciclerView(R.id.frag_promociones_recyclerView)
                    .withEmptyList(R.id.frag_promociones_empty_list)
                    .build();
            break;
        case 2: // Separador
            break;
        case 3: // Settings
            Toast.makeText(getActivity().getApplicationContext(), "Click en Settings", Toast.LENGTH_SHORT).show();
            break;
        case 4: // Logout
//          Toast.makeText(getActivity().getApplicationContext(), "Click en Logout", Toast.LENGTH_SHORT).show();
            logOut();
            break;
        default:
            Toast.makeText(getActivity().getApplicationContext(), "Fragmento no establecido", Toast.LENGTH_SHORT).show()
            break;
    }

    return fragmentToLoad;
}
```

Ilustración 29 - Ejemplo método getFragmentToLoad en CustomerLibrary

El fragmento cargado se determina según la posición seleccionada en el **Navigation Drawer**. En la imagen anterior, también se puede apreciar cómo los fragmentos implementan el **patrón de diseño Builder**, que es un **patrón de creación** extensamente conocido.

FRAGMENTS

LOGIN Y REGISTRO

Para comenzar el apartado de fragmentos, vamos a empezar viendo dos de los principales fragmentos que se encontrará cualquier usuario de la aplicación, que son el fragmento de login y el del registro.

Ilustración 30 - Fragments de Login y Registro

Estos dos fragmentos son muy parecidos, ambos son formularios, y en lo que más se diferencian es en el número de campos que tiene cada uno. Ambos contienen una clase llamada **MyTextWatcher** que se utiliza para validar dinámicamente los campos del formulario.

CUSTOMER

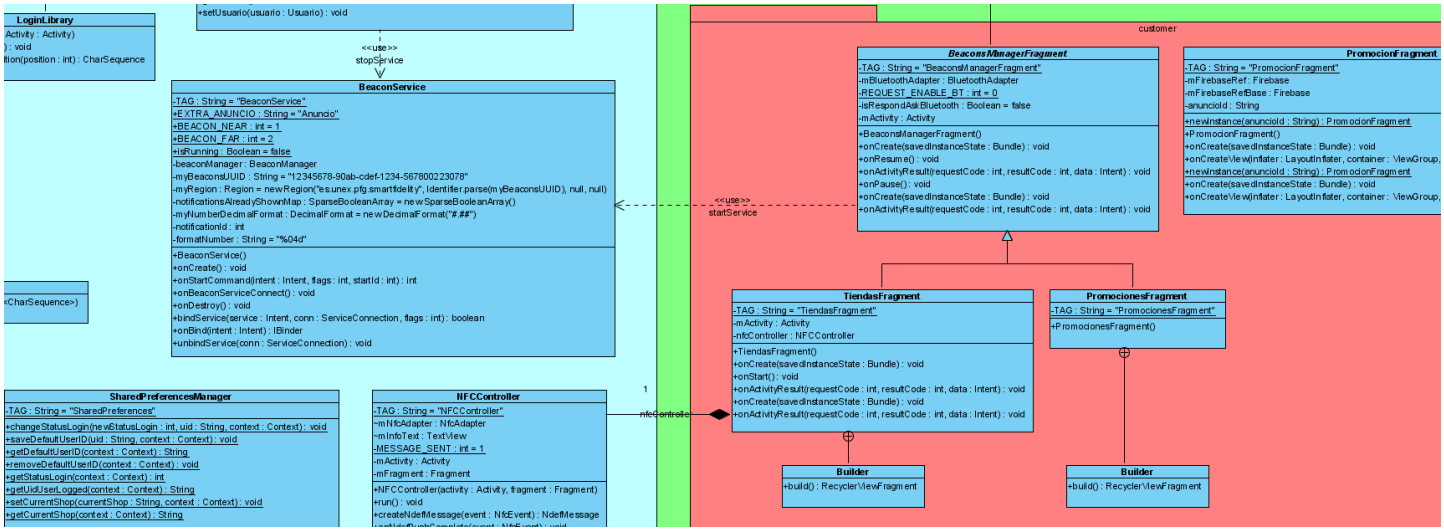


Ilustración 31 - Fragments para Customers

En la imagen anterior vemos tres clases bastante importantes dentro del proyecto, que son: **BeaconService, NFCController** y **SharedPreferencesManager**.

Por un lado tenemos el **SharedPreferencesManager**, que es una clase que tiene métodos estáticos para acceder a las **SharedPreferences** de Android desde cualquier parte de la aplicación para, por ejemplo, leer el Uid del usuario logueado.

Otro elemento importante es el **NFCController,** este elemento permite el envío de mensajes mediante **NFC**. Esta clase es utilizada desde el fragmento **TiendasFragment,** que es el fragmento principal que se le carga a un usuario, y desde donde se le permite registrar una compra.

Por otro lado tenemos el **BeaconService,** esta clase implementa un servicio Android y su función es detectar Beacons y mostrarle al usuario notificaciones cuando los detecte (según la distancia muestra un tipo de notificación u otra). Como podemos ver en el diagrama, este servicio se inicia en la clase abstracta **BeaconsManagerFragment**, de la cual heredan **TiendasFragment** y **PromocionesFragment**, y este servicio es parado desde **MainActivity** (esto es necesario para que si se mata la aplicación no se quede corriendo el servicio). Con esta estructura que hemos mostrado actualmente, si en un futuro quisiésemos añadir otro fragmento nuevo que utilice el servicio de Beacons, simplemente tendríamos que hacer que extendiese **BeaconsManagerFragment**, y únicamente con esta simple acción tendríamos un fragmento que utilizaría el servicio de Beacons. De igual modo, si quisiésemos que alguno de los fragmentos que tenemos actualmente utilizando el servicio de Beacons no lo utilizase, quitaríamos su extensión y dejaría de utilizarlo.

Ya he nombrado en el párrafo anterior tres fragmentos que se están utilizando dentro del paquete **Customer.** Este paquete está dentro del paquete **Fragments** y contiene los fragmentos que se están utilizando actualmente para los clientes. Dentro de este fragmento tenemos: **BeaconsManagerFragment, TiendasFragment, PromocionesFragment y PromocionFragment.**

Comenzando con **BeaconsManagerFragment,** este fragmento es una clase abstracta que utiliza el servicio de **Beacons** y facilita el uso de este servicio a otros fragmentos simplemente extendiendo esta clase. Su principal función es la de comprobar que está encendido el **Bluetooth**, si no lo está, mostrar un mensaje al usuario para preguntarle si quiere que lo activemos, y cuando esté todo correcto iniciar el servicio de **Beacons**. Otra de las funciones de esta clase es la de **poner el servicio de Beacons en modo ahorro de energía** cuando la aplicación pasa a un segundo plano (el usuario cierra la App). Este modo, lo que hace es **reducir la frecuencia de escaneo de Beacons reduciendo, por tanto, el consumo de batería.** Esta clase también extiende de **RecyclerViewFragment** que como explicaré en el siguiente apartado, simplemente se encarga de preparar el fragmento para la carga de una **RecyclerView.**

TiendasFragment es el fragmento principal que se muestra a los clientes al abrir la aplicación y desde donde ellos pueden registrar una compra y ver información de sus tiendas (les permite ver los puntos que tienen en cada tienda) mediante una **RecyclerView** (concretamente una EmtpyRecyclerView que es una extensión propia de la **RecyclerView** de Android y que busca solventar el problema de que la **RecyclerView** de Android no tiene opción de añadir una pantalla para un **EmptyCase** como sí tenía la antigua **ListView**).

PromocionesFragment es un fragmento que muestra una EmtpyRecyclerView con las promociones que hay actualmente publicadas en el sistema.

PromocionFragment es el fragmento que se carga para mostrar el detalle de una promoción a la que ha hecho clic el usuario en una notificación (mostrada por el **BeaconService**).

ADMIN

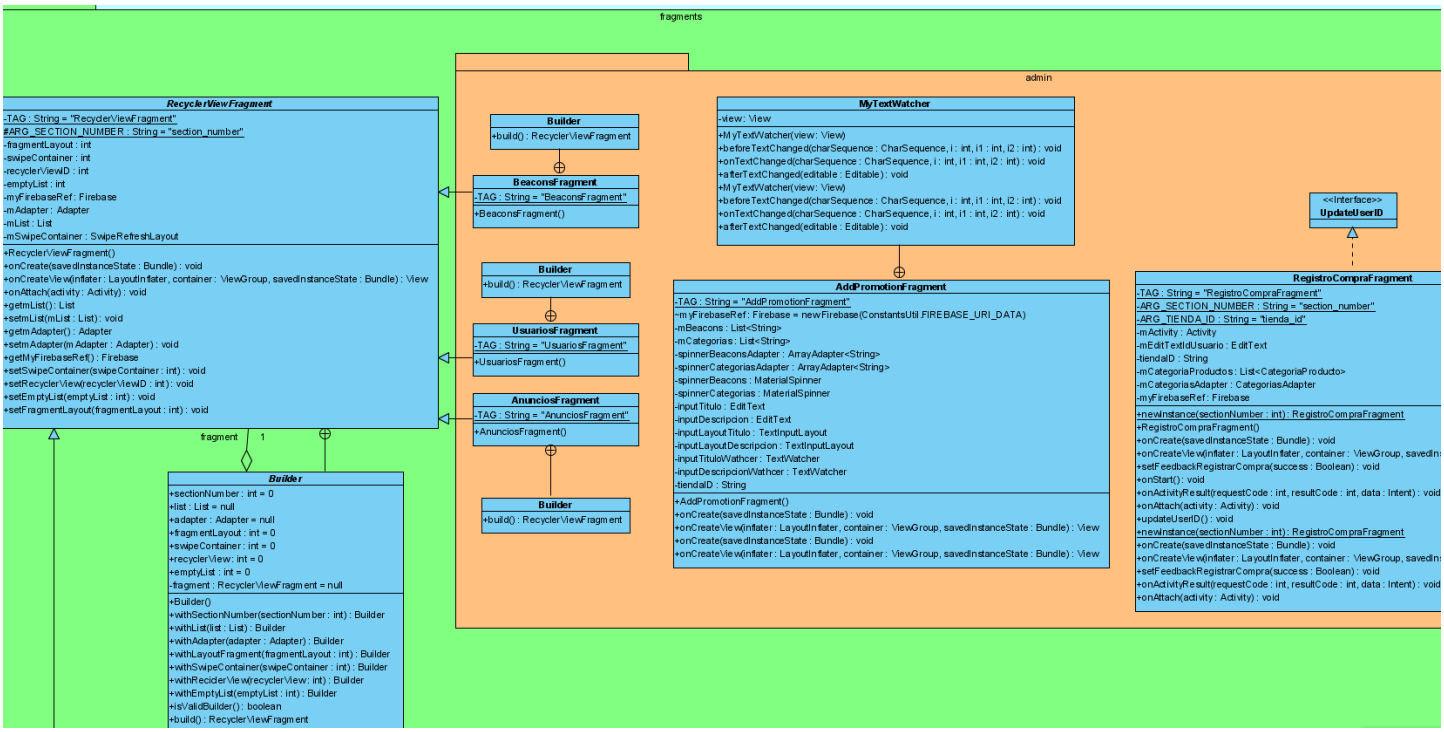


Ilustración 32 - Fragment para Admins

En la anterior imagen podemos ver que tenemos por un lado una clase llamada **RecyclerViewFragment**, la cual ya se ha nombrado en el punto anterior, y por otro lado tenemos una serie de clases dentro de un paquete llamado **Admin**. Este paquete contiene los fragmentos que se están utilizando en la interfaz que tiene a su disposición un usuario administrador.

RecyclerViewFragment es un fragmento que se encarga de configurar una **RecyclerView**, que en este caso es una EmtpyRecyclerView, también se encarga de configurar el SwipeToRefresh, que es un nuevo patrón de interacción añadido en **Material Design** para actualizar una lista, y el Floating Action Button (FAB). Además, implementa el **Patrón de diseño** Builder para facilitar su creación. Debido a que esta clase es abstracta, cualquier clase que extienda esta clase tendrá a su disposición todas las ventajas comentadas anteriormente.

Como vemos en la imagen, un usuario administrador tiene los siguientes fragmentos:

- **BeaconFragment** → Se utiliza para mostrar los Beacons que tiene la tienda. Esta clase extiende **RecyclerViewFragment** para tener a su disposición los elementos que implementa este fragmento.

- **UsuariosFragment** → Se utiliza para mostrar los usuarios de la tienda. También extiende **RecyclerViewFragment.**

- **AnunciosFragment** → Muestra una lista con los anuncios que hay creados actualmente para la tienda. También extiende **RecyclerViewFragment.**

- **AddPromotionFragment** → Se utiliza para permitir al usuario crear una nueva promoción (Anuncio) asociado a la tienda y a un Beacon dentro de esta. Como vemos en la imagen, este fragmento contiene una clase llamada **MyTextWatcher.** Esta clase se utiliza para realizar las **validaciones dinámicas** en los campos del formulario de añadir promoción.

- **RegistroCompraFragment** → Se utiliza como fragmento principal para el usuario administrador y le permite registrar una compra mediante NFC, lectura de código QR o introducción manual del identificador de usuario. En este fragmento se muestra también una lista de categorías de productos que vende la tienda y que se utilizarán para calcular el número de puntos que se le entrega al usuario por realizar esa compra (cada categoría tiene asociados una serie de puntos). Como se puede ver en la imagen, este fragmento implementa una interfaz llamada **UpdateUserId**. Esto se utiliza para tener un **callback** para actualizar el campo de identificador de usuario cuando se recibe la respuesta mediante **NFC**.

Un dato a tener en cuenta, es que los **Builders** de los fragmentos **BeaconFragment, UsuariosFragment y AnunciosFragment** extienden el **Builder** definido en **RecyclerViewFragment** y sobrescriben el método **Build** para construir el objeto **RecyclerViewFragment** con el tipo de fragmento que queremos utilizar en cada caso (este fragmento es el que utiliza el **Builder** para rellenarlo y devolverlo en el método **Build**). Lo mismo pasa con los fragmentos **TiendasFragment y PromocionesFragment** del anterior apartado. Esto nos permite compartir el mismo **Builder** y reducir así las duplicidades en el código. La siguiente imagen es un ejemplo del **Builder** utilizado en **AnunciosFragment**:

```
public class AnunciosFragment extends RecyclerViewFragment {
    private static final String TAG = "AnunciosFragment";

    public static class Builder extends RecyclerViewFragment.Builder {
        @Override
        public RecyclerViewFragment build() {
            super.setFragment(new AnunciosFragment());
            return super.build();
        }
    }

    public AnunciosFragment() {
    }
```

Ilustración 33 - Extensión del patrón Builder en AnunciosFragment

VIEW ADAPTERS

En este apartado vamos a ver la parte del diagrama de clases cuya función es la de crear las vistas para las distintas listas que hay en la aplicación. Estas clases son adaptadores, aunque no están directamente relacionados con el patrón de diseño Adapter, sino más bien, se encargan de construir las vistas que van en las listas, en nuestro caso son **EmptyRecyclerView**, con los datos que hay disponibles para pintar y con el diseño en XML que hayamos elegido para las vistas.

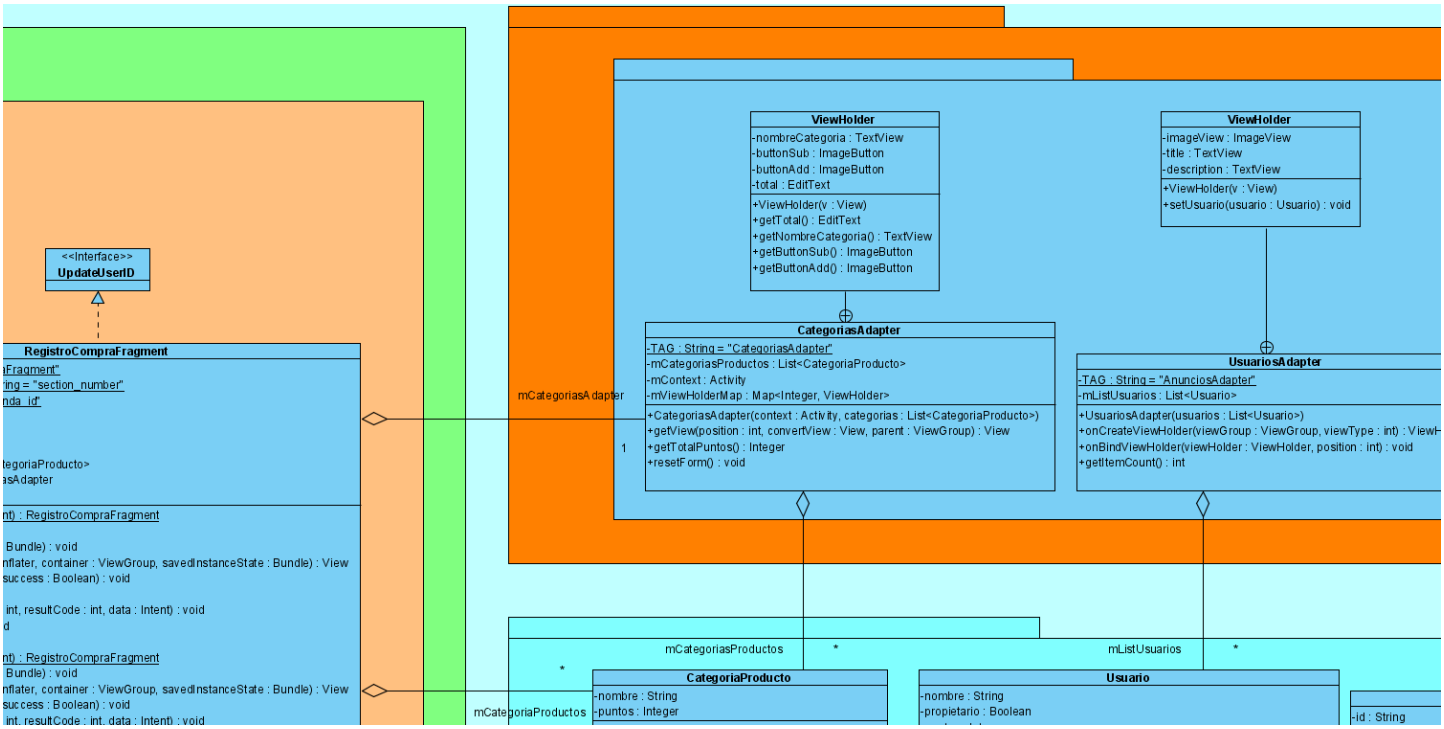


Ilustración 34 - View Adapters I

Comenzamos este apartado con los dos primeros adaptadores que son **CategoriasAdapter** y **UsuariosAdapter.**

El primero (**CategoriasAdapter)** vemos que se utiliza en **RegistroCompraFragment**, y que a su vez utiliza un objeto del modelo de datos llamado **CategoriaProducto.** Se utiliza en **RegistroCompraFragment** porque, como dijimos en el apartado anterior, este fragmento muestra una lista de categorías que hay en la tienda al dependiente de la misma, y para mostrar los datos de las categorías es necesario el uso de este adaptador, que se encarga de coger los datos de las categorías de la tienda y devolver la vista renderizada para cada elemento de la lista. Como vemos, el uso de **CategoriaProducto** es porque tanto el adaptador, como el fragmento, tienen una lista de categorías de productos que hay para esa tienda. En realidad el que tiene la información es el fragmento y se la pasa al adaptador para que pueda renderizar correctamente las vistas para la lista.

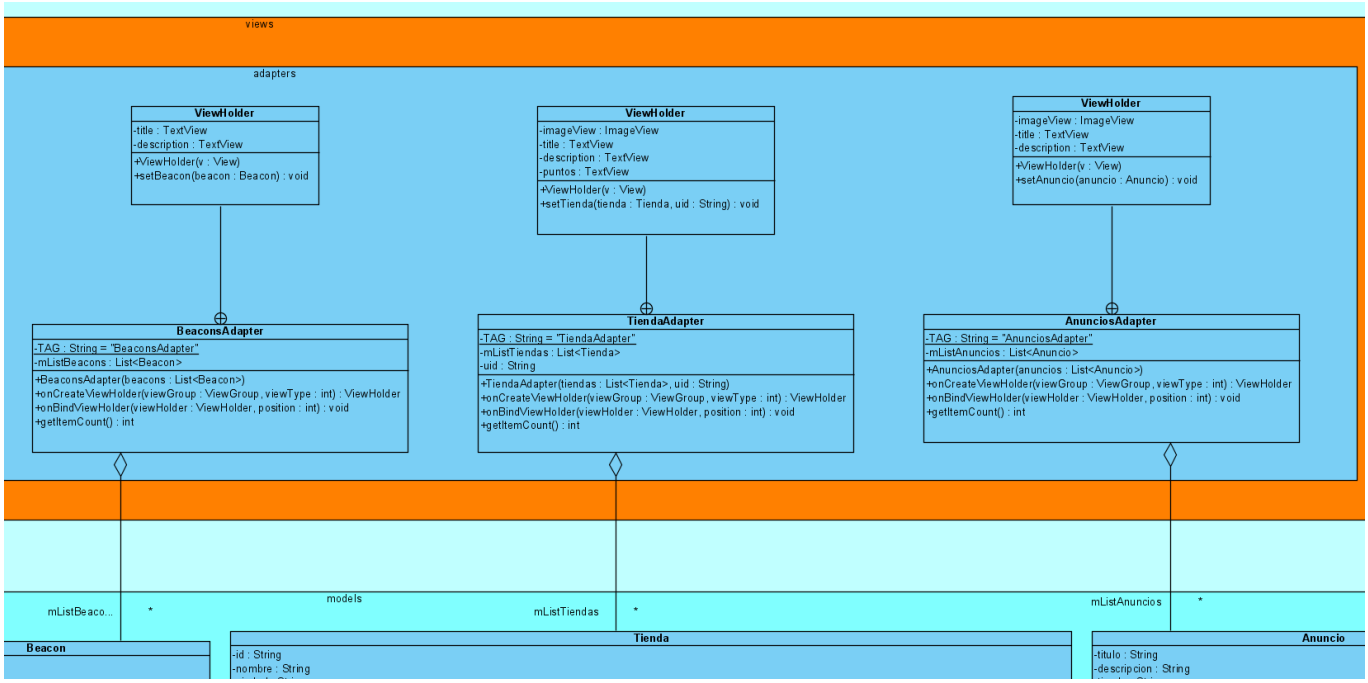


Ilustración 35 - View Adapters II

UsuariosAdapter es un adaptador utilizado para la lista que se muestra en **UsuariosFragment**. Este adaptador tiene el mismo objetivo que el anterior, que es renderizar cada una de las vistas de la lista con los datos de los usuarios que le pasa el fragmento. Para hacer esta tarea posible, es necesario que mantenga una lista con los datos de los usuarios y por eso tiene esa relación con la clase **Usuario** del modelo de datos.

Seguimos con el resto de adaptadores que existen en esta aplicación. En la imagen anterior podemos ver que tenemos tres adaptadores más, que son: **BeaconAdapter, TiendaAdapter y AnuncioAdapter.**

BeaconsAdapter se utiliza para renderizar las vistas que se utilizan en la lista de **Beacons** que puede ver un usuario administrador en el fragmento **BeaconsFragment**.

TiendaAdapter se utiliza para renderizar las vistas que se utilizan en la lista de **Tiendas** que se les muestran a los clientes en el fragmento **TiendasFragment.**

Por último, **AnunciosAdapter** es utilizado para renderizar las vistas de los anuncios que se muestran en la lista que contiene el fragmento **AnunciosFragment**, que se utiliza para mostrarle los anuncios que tiene establecidos un usuario administrador en su tienda, y también se utiliza en el fragmento **PromocionesFragment**, que se utiliza en la interfaz de la parte de clientes para poder ver las promociones disponibles actualmente.

Como podemos ver en las imágenes anteriores, todos los adaptadores contienen una clase llamada **ViewHolder**, esta clase se utiliza para implementar el patrón **ViewHolder**, que es un patrón que se utiliza **para conseguir un mayor rendimiento en el pintado de listas y en la interacción del usuario con ellas**. Básicamente su objetivo es reducir el número de llamadas a los métodos que buscan las referencias a elementos de una vista (la más típica es **findViewById**), encapsulando estas referencias en un objeto (**ViewHolder**).

MODEL

Las clases del modelo tienen como objetivo modelar los distintos elementos de los que se compone la aplicación, y también son utilizados por **Firebase** para **mapear estos objetos a la BD**. Una cosa que hay que tener en cuenta de estas clases es que Firebase utiliza los constructores para el mapeo de objetos, por lo que, si algún atributo es opcional, será necesario tener dos constructores distintos, uno sin este atributo y otro con este atributo, y así **Firebase** podrá utilizar el que más le convenga en cada caso para mapear los documentos **JSON** que almacena en la BD a objetos del modelo.

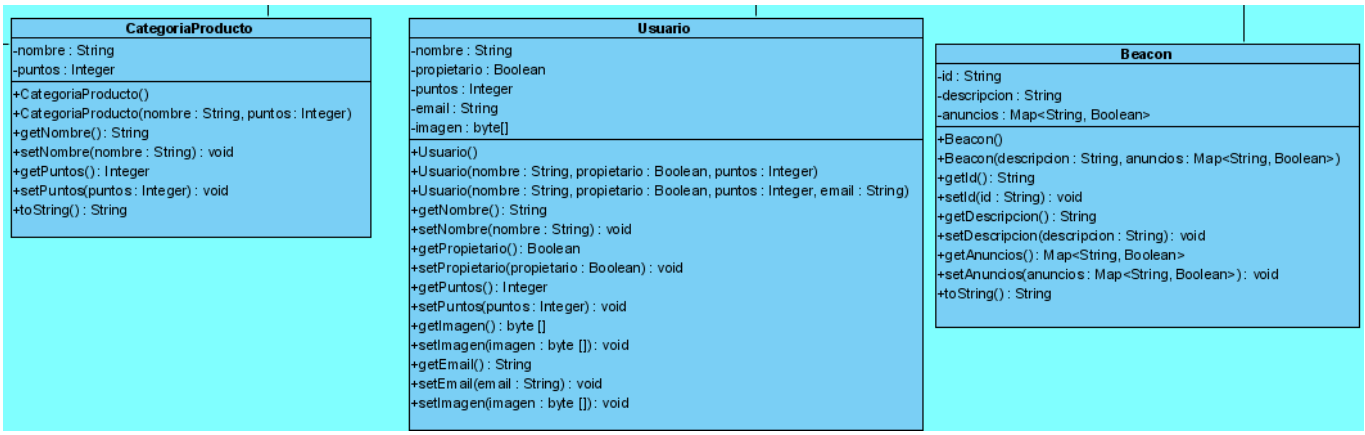


Ilustración 36 - Objetos del modelo de datos I

Como podemos ver en la imagen anterior, la clase **Usuario** tiene definidos tres constructores: uno sin parámetros, que es necesario para Firebase, uno con todos los parámetros obligatorios, y otro añadiendo el email, que es considerado un parámetro opcional.

Las otras dos clases tienen constructores con todos los parámetros, ya que todos son obligatorios.

La siguiente imagen muestra las otras dos clases que faltan del **modelo de datos**, que son: **Tienda y Anuncio.** Como podemos ver, **Tienda** también tiene un parámetro opcional que son categorías, ya que una tienda puede inicialmente ser creada sin las categorías. Anuncio, sin embargo, al igual que las dos clases de la imagen anterior, solo tiene un tipo de constructor con parámetros, el cual es utilizado por **Firebase** para mapear los objetos.

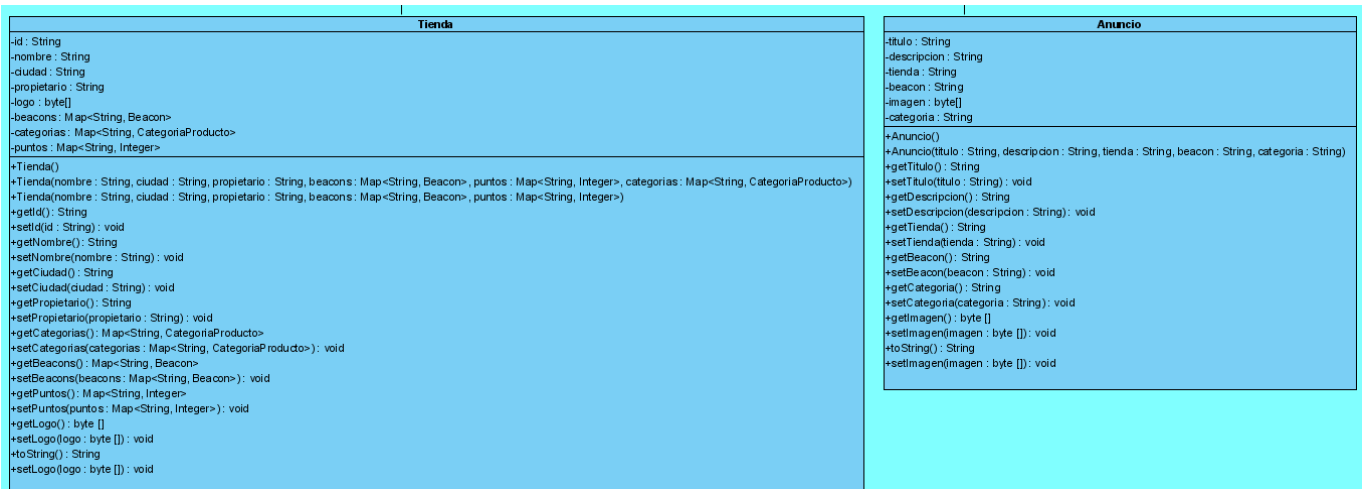


Ilustración 37 - Objetos del modelo de datos II

5.3. DISEÑO E IMPLEMENTACIÓN

5.3.1. ORGANIZACIÓN DEL PROYECTO

Una cosa que considero fundamental en todos los proyectos que hago es la organización del código y la definición clara de la nomenclatura a utilizar en las diversas partes (clases, estilos, nombres de ficheros XML, identificadores...). Poner unas reglas de nomenclatura y organización para todo el proyecto es algo costoso y hay que planificarlo muy bien para que, en el futuro, el proyecto se pueda mantener fácilmente.

Debido a que esto no es una tarea sencilla, se realizó una búsqueda en la red de reglas que se utilizan normalmente en los desarrollos Android, y se encontró el siguiente enlace donde se recogen una serie de buenas prácticas para desarrollos Android que me pareció muy interesante y que he aplicado en este proyecto. El enlace es el siguiente: Android Best Practices.

En este proyecto se definen varias reglas, de las cuales me gustaría destacar las que yo he aplicado y cómo lo he hecho:

- **Java packages architecture** → Esto para mí fue lo más importante, la organización del código en paquetes facilita mucho el desarrollo y el mantenimiento. En este punto, ellos nos proponen una estructura que es la que yo he elegido para mi proyecto, aunque he añadido algún nivel más de paquete (en fragments). Como podemos ver en la siguiente imagen cada fichero está localizado donde debe y los ficheros principales se encuentran en el paquete principal.

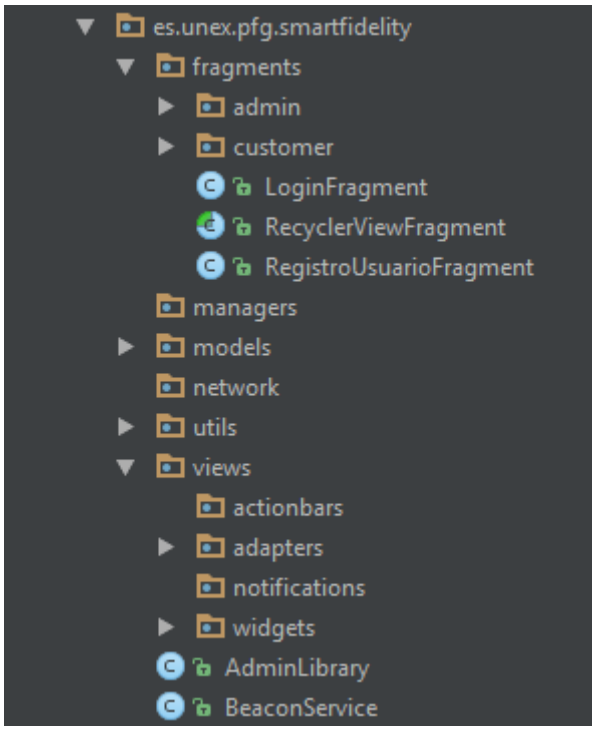


Ilustración 38 - Estructura de paquetes

- **Use Fragments to represent a UI screen** → Este es uno de los aspectos que siempre he seguido en los proyectos Android. Me parece que se gana mucha versatilidad representando la interfaz de usuario con fragmentos en lugar de actividades. Las actividades son algo más pesadas y no son reutilizables, por eso es preferible utilizar fragmentos que pueden ser reutilizados y son más fáciles de cargar por el sistema. Como veremos en el siguiente apartado, este proyecto está completamente desarrollado en base a fragmentos y solo tiene una interfaz general que hace las funciones de controlador.

- **Use Activities just to manage Fragments** → Este punto está estrechamente relacionado con el anterior y lo que viene a decir es que las actividades se tienen que dedicar exclusivamente a manejar fragmentos, evitando que sean un monolito de código que después es muy difícil de cambiar.

- **Layout XMLs are code, organize them well** → En este punto lo que yo he hecho es tomar una serie de normas de nomenclatura para poder organizar los layouts. Una de las cosas que hago es nombrar el fichero según el tipo de layout que representa, por ejemplo podemos tener los siguientes nombres: **fragment_anuncios.xml y view_anuncio_item.xml**. El primero contiene la UI para el fragmento en el que se muestran la lista de anuncios, mientras que el segundo fichero es una vista que contiene la definición de la UI de un anuncio en la lista. Haciendo esto, si quisiésemos cambiar la forma en la que se muestra un anuncio se cambiaría solo la vista y si queremos mostrar algún elemento más a parte de la lista, por ejemplo un botón, lo haríamos en la definición del fragmento. También he definido una serie de reglas para nombrar cada elemento dentro de un fichero, haciendo que estos sean lo más descriptivos posibles y evitando duplicidades en el código y facilitando la búsqueda. Para ello lo que hago es que comiencen por el nombre del fichero y luego por algo descriptivo:

```
<!-- Vista que se utiliza como EmptyView para la EmptyRecyclerView -->
<LinearLayout
    android:id="@+id/frag_anuncios_empty_list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

Ilustración 39 - Naming en vistas XML

- **Use styles to avoid duplicate attributes in layout XMLs** → Esto es algo muy útil para conseguir el **DRY (Don`t Repeat Yourself)**. Consiste en definir estilos para ciertas partes de los diseños que se suelen repetir y así reutilizar estos diseños. En mi caso por ejemplo lo he utilizado para el diseño de los spinners, he definido un estilo en el fichero styles.xml llamado MaterialSpinnerStyle en el que defino el diseño general que quiero para estos spinners y luego lo utilizo en su definición.

```
<fr.ganfra.materialspinner.MaterialSpinner
    android:id="@+id/frag_add_promocion_beacon"
    style="@style/MaterialSpinnerStyle"
    android:layout_width="wrap_content"
```

Ilustración 40 - Reutilización de estilos

- **Keep your colors.xml short and DRY, just define the palette** → Esto también busca conseguir el DRY evitando que pongamos colores *hardcodeados* sino que definamos los colores en el fichero de colores evitando estas duplicidades y facilitando el mantenimiento.

```
    android:layout_height="match_parent"
    android:background="@color/theme_highlight"
    android:fitsSystemWindows="true"
```

Ilustración 41 - Definición de colores

- **Also keep dimens.xml DRY, define generic constants** → Este punto es igual que el anterior pero con las dimensiones.

```
    android:layout_marginBottom="@dimen/cardview_margin_top_bottom"
    android:layout_marginLeft="@dimen/cardview_margin_left_right"
    android:layout_marginRight="@dimen/cardview_margin_left_right"
    android:layout_marginTop="@dimen/cardview_margin_top_bottom"
```

Ilustración 42 - Definición de dimensiones

5.3.2. PATRONES DE DISEÑO UTILIZADOS

STRATEGY

El **patrón de diseño Strategy** está clasificado como un **patrón de diseño de comportamiento [51, 52].** Se clasifica como un patrón de comportamiento porque **determina cómo se realiza alguna acción de forma dinámica simplemente cambiando la instanciación del objeto** (en nuestro caso determina cómo se realiza la carga del fragmento). La siguiente imagen muestra cómo se estructura este patrón mediante un diagrama UML.

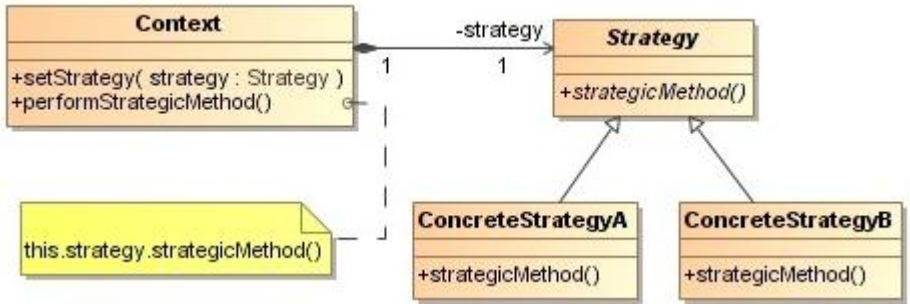


Ilustración 43 – Diagrama de clases del patrón de diseño Strategy

La gran ventaja de este patrón es que nos permite reutilizar el código cambiando solamente aquellas partes que son distintas según un determinado contexto. Por ejemplo, si estamos haciendo un conversor de video en el que permitiremos varios formatos de entrada y de salida, si utilizamos este patrón en nuestro programa, podremos tener varias clases para interpretar el video de entrada y pasarlo a un formato estándar y otras clases para pasar de ese formato estándar a uno concreto de salida. El código principal de este programa será el mismo para todas las transformaciones, y solo cambiará la forma en la que se interpreta y se convierte el video según la elección del usuario. Además de esto, si en un futuro queremos añadir un nuevo formato, será tan sencillo como crear una clase que implemente las interfaces para interpretar y/o convertir un video.

Este patrón se ha utilizado en este proyecto en la **MainActivity** para seleccionar la librería (**LoginLibrary**, **CustomerLibrary** o **AdminLibrary**) según el contexto en el que se encuentre la aplicación. Estas implementaciones concretas mostrarán al usuario unas opciones u otras según ese contexto.

En javacodegeeks.com[24] se puede ver más en detalle en qué consiste este patrón de diseño y un ejemplo de su implementación.

BUILDER

El **patrón de diseño Builder** está clasificado como un **patrón de creación [53, 54]**. Su **objetivo** es abstraer el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

La siguiente imagen muestra un diagrama de clases de cómo se utiliza este patrón.

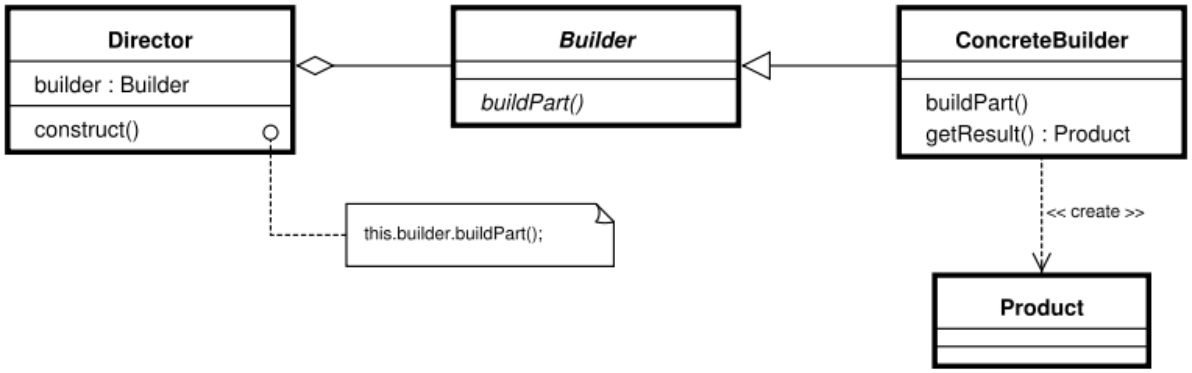


Ilustración 44 - Diagrama de clases del Patrón de diseño Builder

Este patrón de diseño se ha utilizado en la **clase abstracta RecyclerViewFragment**, y al ser abstracta esta características es heredada por todas las clases que extienden esta clase, que en este proyecto son todas los fragmentos que utilizan una **RecyclerView**.

En javacodegeeks.com[25] se puede encontrar una explicación más detallada junto con un ejemplo de implementación.

[24] Strategy Design Pattern <http://www.javacodegeeks.com/2015/09/strategy-design-pattern.html> [Disponible a 31 de enero de 2016]

[25] Builder Design Pattern <http://www.javacodegeeks.com/2015/09/builder-design-pattern.html> [Disponible a 31 de enero de 2016]

VIEWHOLDER

ViewHolder es un patrón que es ampliamente utilizado en el desarrollo de aplicaciones Android [50]. Se utiliza para conseguir un mayor rendimiento en el pintado de listas y en la interacción del usuario con ellas, ya que cuando un elemento de la lista no está visible, éste se elimina del árbol de vistas que mantiene Android, y cuando ese elemento vuelve a ser requerido, es necesario volver a pintarlo. Debido a esto, el patrón **ViewHolder** lo que busca es reducir el número de llamadas a los métodos que buscan las referencias a elementos de una vista (la más típica es **findViewById**), encapsulando estas referencias en un objeto (**ViewHolder**).

Como vemos, la idea es muy sencilla y las mejoras son significativas, por eso es ampliamente utilizado y conocido en el desarrollo de aplicaciones Android. A continuación muestro un pequeño ejemplo de un **ViewHolder** utilizado para almacenar las referencias a los elementos de la vista de un **Beacon**.

```
/**
 * Provide a reference to the type of views that you are using (custom ViewHolder)
 */
public static class ViewHolder extends RecyclerView.ViewHolder {
    private TextView title, description;

    public ViewHolder(View v) {
        super(v);
        // Define click listener for the ViewHolder's View.
        v.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.i(TAG, "Element " + getPosition() + " clicked.");
            }
        });

        title = (TextView) v.findViewById(R.id.view_beacon_titleTextView);
        description = (TextView) v.findViewById(R.id.view_beacon_descriptionTextView);
    }

    public void setBeacon(Beacon beacon) {
        title.setText(beacon.getId());
        description.setText(beacon.getDescripcion());
    }
}
```

Ilustración 45 – Ejemplo del Patrón ViewHolder

Como podemos ver, simplemente tenemos en este caso dos atributos privados dentro del objeto que representan los elementos que tiene esta vista, en este caso son dos **TextView**.

El constructor de esta clase recibe la vista que va a almacenar y lo que hace es buscar la referencia a cada elemento de la vista y lo almacena en los atributos. Como vemos, también se encarga de establecer un *listener* para cuando el usuario haga clic en esta vista.

También vemos que estas clases pueden tener métodos para pintar datos en la vista. En este caso tenemos un método llamado **setBeacon** que recibe un **Beacon** y establece sus datos en la vista.

Un ejemplo de utilización de este patrón podría ser el siguiente:

```
@Override
public ViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
    // Create a new view.
    View v = LayoutInflater.from(viewGroup.getContext())
            .inflate(R.layout.view_beacon_item, viewGroup, false);

    return new ViewHolder(v);
}

// Replace the contents of a view (invoked by the layout manager)
@Override
public void onBindViewHolder(ViewHolder viewHolder, final int position) {
    Log.i(TAG, "Element " + position + " set.");

    // Get element from your dataset at this position and replace the contents of the view
    // with that element
    viewHolder.setBeacon(mListBeacons.get(position));
}
```

Ilustración 46 - Ejemplo de uso del patrón ViewHolder

La imagen anterior sigue siendo en la clase **BeaconAdapter**. Podemos ver cómo en el método **onCreateViewHolder** se recibe una vista, la cual se "infla" con la vista del **Beacon** y se le pasa a la clase **ViewHolder** para crear un objeto y devolverlo.

El método **onBindViewHolder** recibe como parámetro el **ViewHolder** que se quiere cargar y la posición que representa dentro de la lista. Con estos datos lo que se hace es obtener el objeto del modelo de datos y cargar su información en la vista, mediante el método **setBeacon.**

Este patrón es utilizado en todos los adaptadores de la aplicación.

TEMPLATE METHOD

Este patrón de diseño esta clasificado como un **patrón de diseño de comportamiento**. Su objetivo es delegar en las clases concretas parte de la logica de un algoritmo o proceso.

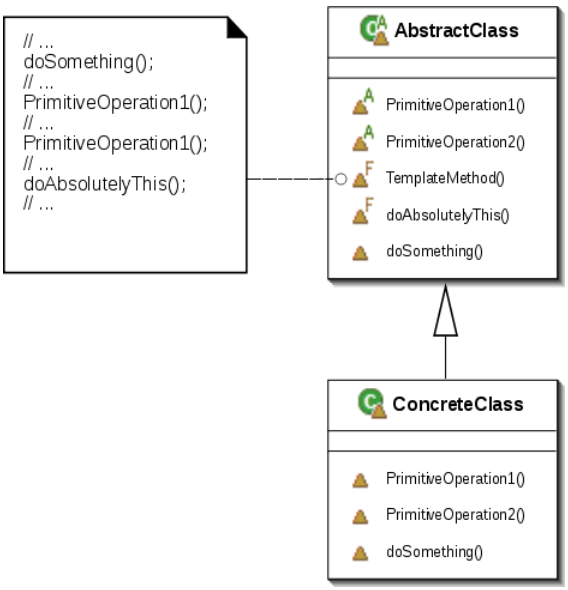


Ilustración 47 - Template Method

Este patrón de diseño es muy común y muchas veces es utilizado sin conocimiento de que es un patrón de diseño. Es ampliamente utilizado por muchos **Frameworks,** un claro ejemplo podría ser el ciclo de vida de una actividad de Android. El **ciclo de vida de una actividad Android es un algoritmo que consiste en realizar una serie de llamadas a unos metodos en un orden concreto** y en el que **Android** ya realiza unas operaciones concretas en cada una de estas operaciones, pero nosotros como desarrolladores podemos sobreescribir alguno de estos metodos y para realizar ciertas operaciones. El caso más comun es el método **onCreate,** que es sobreescrito en todas las actividades para realizar la carga de la vista que va a mostrar la actividad y para realizar el setup de la aplicación. La siguiente imagen muestra como es **ciclo de vida de una actividad en Android**.

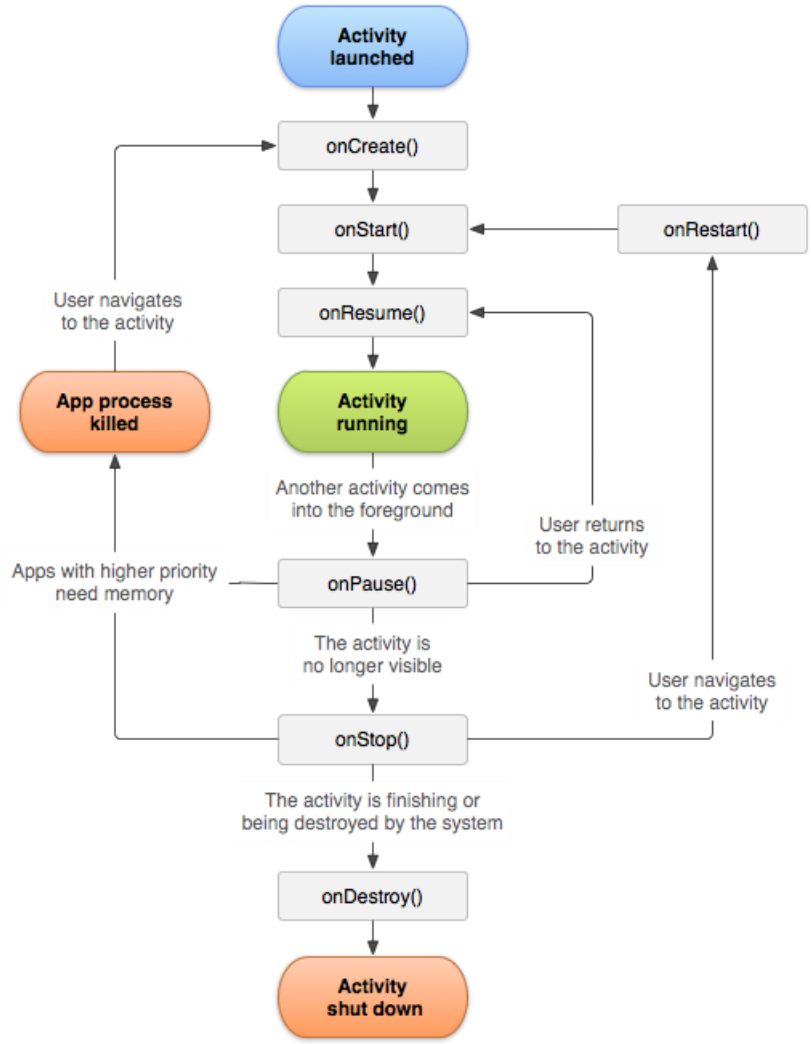


Ilustración 48 - Template Method en el ciclo de vida de una Actividad Android

Este patrón de diseño se utilizan en la clase abstracta **MainLibraryWithDrawer,** en el método **loadFragment** se llama a un método abstracto llamado **getFragmentToLoad** que devuelve el fragmento que debe ser cargado y que es implementado por las clases concretas que extienden de esta clase abstracta. Tambien, en esta misma clase se utiliza para en la configuracion del **NavigationDrawer** que se hace en el método privado **getDrawer** donde las opciones que va a tener el drawer se establecen por un método abstracto llamado **getDrawerItems** que las clases abstractas que extienden esta clase son las encargadas de establecer.

En otra clase abstracta donde se hace uso de este patrón de diseño es en **RecyclerViewFragment**. Esta clase tiene un método privado llamado **setSwipeToRefresh** que se encarga de configurar el swipe to refresh y que en la acción principal de este elemento (refrescar una vista) llama a un método abstracto llamado **updateList** que es implementado por todas las clases que extienden de esta clase y donde pueden definir que es lo que quieren que suceda en cada caso cuando el usuario solicite refrescar la **RecyclerView**.

```
private void setSwipeToRefresh(View rootView) {
    mSwipeContainer = (SwipeRefreshLayout) rootView.findViewById(swipeContainer);
    // Configure the refreshing colors
    mSwipeContainer.setColorSchemeResources(android.R.color.holo_blue_bright,
            android.R.color.holo_green_light,
            android.R.color.holo_orange_light,
            android.R.color.holo_red_light);
    // Setup refresh listener which triggers new data loading
    mSwipeContainer.setOnRefreshListener(() -> {
            // Your code to refresh the list here.
            // Make sure you call mSwipeContainer.setRefreshing(false)
            // once the network request has completed successfully.
            Log.d(TAG, "En onRefresh");
            updateList();
    });
}
```

Ilustración 49 - setSwipeToRefresh Template Method

```
protected abstract void updateList();
```

Ilustración 50 - Definicion del método abstracto en Template Method

```java
/**
 * @link RecyclerViewFragment#updateList()
 */
@Override
protected void updateList() {
    Log.d(TAG, "En updateList");

    String tiendaID = SharedPreferencesManager.getCurrentShop(getActivity().getApplicationContext());
    getMyFirebaseRef().child("Tiendas").child(tiendaID).child("beacons").addListenerForSingleValueEvent(new ValueEventListener()

        @Override
        public void onDataChange(DataSnapshot dataSnapshot) {
            Log.d(TAG, "En onDataChange de updateList");

            // Procesamos el resultado
            Iterator<DataSnapshot> iterator = dataSnapshot.getChildren().iterator();
            Beacon beaconRead;
            DataSnapshot beaconDataSnapshot;
            List mList = getmList();
            mList.clear();
            while (iterator.hasNext()) {
                beaconDataSnapshot = iterator.next();
                beaconRead = beaconDataSnapshot.getValue(Beacon.class);
                // Establezco el id, para ello utilizo la clave
                beaconRead.setId(beaconDataSnapshot.getKey());
                mList.add(beaconRead);
            }

            updateRecyclerView();
        }

        @Override
        public void onCancelled(FirebaseError firebaseError) {
            Log.d(TAG, "No se ha podido obtener los Beacons");
            cancelRefreshingAnimation();
        }
    });
}
```

Ilustración 51 - Ejemplo de implementación del método updateList

Como se puede ver en la imagen anterior, lo que se hace en el método **updateList** es consultar la información que se está mostrando al usuario en la **RecyclerView** (en este caso son **Tiendas**) y actualizar la **RecyclerView**. En esa misma imagen vemos que se está haciendo uso del patrón de diseño **Iterator** que es otro patrón de diseño para recorrer una colección de objetos.

5.3.3. LIBRERÍAS

BLUETOOTH ANDROID

FUNCIONALIDAD

En este proyecto se ha utilizado el API que proporciona Android para poder acceder a las características **Bluetooth** de los dispositivos [57]. Como ya se ha comentado anteriormente en este documento, para poder detectar y trabajar con **Beacons**, es necesario tener activado el **Bluetooth** en el dispositivo y además que sea compatible con **Bluetooth Low Energy**.

El uso principal que se ha hecho de la API Bluetooth de Android es para detectar si el dispositivo tiene habilitado el Bluetooth, y en caso de que no esté habilitado, solicitar al usuario que lo habilite. Todo el uso de esta API se encuentra en el fragmento **BeaconsManagerFragment,** que como vimos en puntos anteriores, es el fragmento del que extienden otros fragmentos en los que se quiere detectar **Beacons**.

EJEMPLO DE USO

Lo primero que hay que hacer para poder utilizar esta API, es añadir los permisos necesarios mediante **<uses-permission>** en el fichero **AndroidManifest.xml.** En este caso, también se ha añadido el requisito de que el dispositivo tiene que disponer de Bluetooth para poder instalar esta aplicación mediante el uso de **<uses-feature>**:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-feature android:name="android.hardware.bluetooth"/>
```

Para el uso que necesitamos en esta aplicación de esta API, se han utilizado dos clases: **BluetoothManager** y **BluetoothAdapter,** que debemos importar en la clase donde vayamos a utilizarlas.

```
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothManager;
```

Ilustración 52 - Import de librerías Bluetooth

El **BluetoothManager** se utiliza para acceder al servicio **Bluetooth** de Android y mediante este servicio obtenemos el **BluetoothAdapter.**[26]

```
/*
 * Bluetooth in Android 4.3+ is accessed via the BluetoothManager, rather than
 * the old static BluetoothAdapter.getInstance()
 */
BluetoothManager manager = (BluetoothManager) mActivity.getSystemService(Activity.BLUETOOTH_SERVICE);
mBluetoothAdapter = manager.getAdapter();
```

Ilustración 53 - Obtención de instancia de BluetoothAdapter

Con la instancia de **BluetoothAdapter** será con la que comprobemos si el usuario tiene activo el **Bluetooth** de la siguiente forma[27]:

```
/*
 * We need to enforce that Bluetooth is first enabled, and take the
 * user to settings to enable it if they have not done so.
 */
if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {

    if (isRespondAskBluetooth) {
        Log.e(TAG, "ERROR. El bluetooth es necesario para utilizar la App");
        mActivity.finish();
    }

    //Bluetooth is disabled
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
} else {
    startService();
}
```

Ilustración 54 - Comprobación de estado Bluetooth

[26] Método **onCreate** de **BeaconsManagerFragment**
[27] Método **onResume** de **BeaconsManagerFragment**

Si el usuario no tenía activado el Bluetooth, se le mostrará una pantalla solicitándole que lo active mediante el **Intent** que se puede ver en la imagen anterior. La repuesta de esa pantalla se procesa así[28]:

```
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    Log.i(TAG, "En onActivityResult. requestCode: " + requestCode);
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == REQUEST_ENABLE_BT) {
        isRespondAskBluetooth = true;
        if (resultCode == Activity.RESULT_OK) {
            Snackbar.make(getView(), "Bluetooth activado", Snackbar.LENGTH_LONG).show();
            Log.d(TAG, "Bluetooth activado");
        } else {
            Snackbar.make(getView(), "No Bluetooth", Snackbar.LENGTH_LONG).show();
            Log.d(TAG, "No Bluetooth");
            mActivity.finish();
        }
    }
}
```

Ilustración 55 - Tratamiento de la solicitud de activación Bluetooth

CONCLUSIÓN

El uso de esta librería ha sido bastante sencillo, ya que la **documentación de Android** es bastante buena. Gracias a que está ofrecida directamente por Android, es muy fácil utilizarla y nos permite dar una mejor experiencia al usuario.

[28] Método **onActivityResult** de **BeaconsManagerFragment**

NFC ANDROID

FUNCIONALIDAD

NFC se ha utilizado para permitir la transferencia del identificador de usuario entre el dispositivo del cliente y el del establecimiento. Para acceder a las características **NFC** del dispositivo, se ha utilizado la API que proporciona **Android.**

EJEMPLO DE USO

Lo primero que hay que hacer para poder utilizar las características **NFC** del dispositivo es modificar el fichero **AndroidManifest.xml** para añadir el permiso necesario mediante **<uses-permission>**:

```
<uses-permission android:name="android.permission.NFC"/>
```

Ilustración 56 - Permiso NFC

Para poder recibir los mensajes **NFC** en nuestra aplicación, y que se abra si no está abierta al recibir un nuevo mensaje, es necesario añadir un nuevo **<intent-filter>** a la **MainActivity**, además de cambiar el **launchMode** a **singleInstance** para que si está abierta ya la aplicación, sólo llame al método **onNewIntent** de la **MainActivity.**

```
<activity
    android:name=".MainActivity"
    android:configChanges="orientation|screenSize|keyboardHidden"
    android:label="SmartFidelity"
    android:launchMode="singleInstance">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
    <intent-filter>
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>

        <category android:name="android.intent.category.DEFAULT"/>

        <data android:mimeType="application/es.unex.pfg.smartfidelity"/>
    </intent-filter>
</activity>
```

Ilustración 57 - Intent Filter para recibir mensajes NFC en MainActivity

En este proyecto se ha creado una clase llamada **NFCController** encargada de facilitar la transferencia de mensajes mediante NFC. En esta clase se utiliza la API que nos proporciona **Android** para el uso de NFC. La clase principal que se utiliza para hacer uso de NFC es **NfcAdapter.** Los pasos para enviar un mensaje mediante NFC son:

1. Obtener la instancia de NfcAdapter para la actividad actual.

2. Comprobar que NFC está activado. Si no lo está, solicitar al usuario que lo active abriéndole las opciones de configuración Wireless, que es donde se puede activar NFC. También puede ser que el dispositivo no tenga NFC, por lo que se le informará al usuario en tal caso.

3. Establecer un callback para enviar el mensaje mediante NFC (implementar **NfcAdapter.CreateNdefMessageCallback**)

4. Establecer un callback para ser notificados cuando la transferencia se haya realizado con éxito (implementar **NfcAdapter.OnNdefPushCompleteCallback**)

Los pasos anteriores se realizan en el método *run* de la clase **NFCController:**

```
public void run() {
    Log.d(TAG, "En run");
    // Check for available NFC Adapter
    mNfcAdapter = NfcAdapter.getDefaultAdapter(mActivity);
    if (mNfcAdapter != null && mNfcAdapter.isEnabled()) {
        // Register callback to set NDEF message
        mNfcAdapter.setNdefPushMessageCallback(this, mActivity);
        // Register callback to listen for message-sent success
        mNfcAdapter.setOnNdefPushCompleteCallback(this, mActivity);
        Toast.makeText(mActivity.getApplicationContext(), "Callbacks NFC registrados", Toast.LENGTH_LONG).show();
        Snackbar.make(mFragment.getView(), "Callbacks NFC registrados", Snackbar.LENGTH_LONG).show();
        Log.d(TAG, "Callbacks NFC registrados");
    } else {
        if (mNfcAdapter == null) {
            Toast.makeText(mActivity.getApplicationContext(), "The device not support NFC", Toast.LENGTH_LONG).show();
            Log.e(TAG, "ERROR al obtener el NFCAdapter. El dispositivo no soporta NFC");
        } else {
            Toast.makeText(
                    mActivity.getApplicationContext(),
                    "Please activate NFC and press Back to return to the application!",
                    Toast.LENGTH_LONG
            ).show();
            Log.d(TAG, "The NFC is Disabled");
            mActivity.startActivity(new Intent(android.provider.Settings.ACTION_WIRELESS_SETTINGS));
        }
    }
}
```

Ilustración 58 - Método run de NFCController

La creación del mensaje NFC se hace en el método **createNdefMessage** que hay que implementar. En este método se crea un objeto **NfcRecord,** donde se establece un **mimeType** y el **texto a transferir,** y se encapsula dentro de un **NfcMessage** que son los mensajes que se envían mediante NFC.

```
@Override
public NdefMessage createNdefMessage(NfcEvent event) {
    String uid = SharedPreferencesManager.getUidUserLogged(mActivity.getApplicationContext());
    return new NdefMessage(NdefRecord.createMime("application/es.unex.pfg.smartfidelity", uid.getBytes()));
}
```

Ilustración 59 - Método createNdefMessage de NFCController

Cuando el mensaje se ha enviado, se llama al método **onNdefPushComplete.** En este caso simplemente mostramos un mensaje al usuario para indicarle que el mensaje se ha enviado correctamente.

```
@Override
public void onNdefPushComplete(NfcEvent event) {
    Log.d(TAG, "En onNdefPushComplete");
    // A handler is needed to send messages to the activity when this
    // callback occurs, because it happens from a binder thread
    mHandler.obtainMessage(MESSAGE_SENT).sendToTarget();
}

/**
 * This handler receives a message from onNdefPushComplete
 */
private final Handler mHandler = handleMessage(msg) -> {
        switch (msg.what) {
            case MESSAGE_SENT:
                Snackbar.make(mFragment.getView(), "UID sent correctly", Snackbar.LENGTH_LONG).show();
                Log.d(TAG, "UID sent correctly");
                break;
        }
};
```

Ilustración 60 - Método onNdefPushComplete de NFCController

Para iniciar el envío del mensaje, el fragmento **TiendasFragment** (que es donde le aparece al usuario la opción de registrar compra mediante NFC) crea un objeto **NFCController** y llama al método **run** al pulsar en el botón de registrar compra mediante NFC:

```
buttonNFC.setOnClickListener((v) → {
        Log.d(TAG, "Click en buttonNFC");
        nfcController = new NFCController(mActivity, TiendasFragment.this);
        nfcController.run();
});
```

Ilustración 61 - Ejecución de NFCController

En el receptor, la clase **MainActivity** sobrescribirá el método **onNewIntent**, donde simplemente se establecerá el **Intent** recibido y se procesará.

```
@Override
public void onNewIntent(Intent intent) {
    Log.d(TAG, "En onNewIntent");
    // onResume gets called after this to handle the intent
    setIntent(intent);
    processIntent(intent);
}
```

Ilustración 62 - Método onNewIntent de MainActivity

A continuación se muestra el código utilizado para procesar un nuevo **Intent** que puede ser NFC. Este método lo primero que hace es **comprobar si el Intent recibido es un mensaje NFC,** si lo es, **obtiene el mensaje,** que en nuestro caso será el **UID del usuario**. Se comprueba si el usuario que está logueado es un **usuario Admin,** ya que son los únicos que deberían recibir un UID, si es así, **se guarda el UID leído en las SharedPreferences** y se llama al método **updateUserID** que debería tener implementado el fragmento que se encuentre cargado actualmente en la aplicación. Ese método simplemente lee de las **SharedPreferences** el UID que se acaba de guardar y lo pone como UID de usuario en el **TextView**.

```
void processIntent(Intent intent) {
    Log.d(TAG, "En processIntent con action: " + intent.getAction());
    // Check to see that the Activity started due to an Android Beam
    if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(intent.getAction())) {
        Parcelable[] rawMsgs = intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
        // only one message sent during the beam
        NdefMessage msg = (NdefMessage) rawMsgs[0];
        // record 0 contains the MIME type, record 1 is the AAR, if present
        String uid = new String(msg.getRecords()[0].getPayload());

        int statusLogin = SharedPreferencesManager.getStatusLogin(getApplicationContext());
        if (statusLogin == ConstantsUtil.ADMIN_USER_LOGGED) {
            SharedPreferencesManager.saveDefaultUserID(uid, getApplicationContext());
            if (mLibrary != null) {
                try {
                    // Utilizado para actualizar el UID del fragmento si esta cargado
                    ((UpdateUserID) mLibrary.getFragment()).updateUserID();
                } catch (ClassCastException e) {
                    Log.d(TAG, "El fragmento RegistroCompraFragment no esta cargado");
                }
            }
        } else {
            Log.d(TAG, "El usuario que esta logueado no es ADMIN. Omitido el guardado del UID recibido por NFC");
        }

    } else {
        if (intent.hasExtra(BeaconService.EXTRA_ANUNCIO)) {
            Log.d(TAG, "Deshabilitando sensor de orientacion");
            setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_NOSENSOR);
            fragmentToLoad = PromocionFragment.newInstance(intent.getStringExtra(BeaconService.EXTRA_ANUNCIO));
        }
    }
}
```

Ilustración 63 - Método processIntent en MainActivity

CONCLUSIÓN

La API de NFC ha sido un poco difícil de utilizar, sobre todo para entender cómo funcionaba al principio el tema del envío de mensajes mediante las clases **NfcRecord** y **NfcMessage,** y posteriormente para implementar una solución que fuese fácil de utilizar por parte del usuario y fácil de mantener para los desarrolladores.

También costó bastante controlar el tema de que el usuario puede que no tenga NFC o que no lo tenga activado, y cómo dar la mejor experiencia de usuario posible en estos casos. Para ello, se ha optado por abrirle las opciones de configuración Wireless si no tiene habilitado NFC, mostrándole también un mensaje recordándole que es necesario que lo habilite para poder utilizar esta característica de la aplicación.

Otro punto que costó bastante era el tema de ver cómo podía abrir la aplicación si ésta estaba cerrada y que se estableciese correctamente el UID transferido, y también que no se abriese otra nueva instancia de la aplicación cuando ya estuviese abierta y recibiese un mensaje mediante NFC. Esto se resolvió mediante el cambio del **launchMode** de la actividad a **singleInstance** en el fichero **AndroidManifest.xml**, y gestionando los intents recibidos en el método **onNewIntent**.

Al final, la solución planteada funciona en ambos casos y permite que el uso de esta API sea un poco más sencilla, al tener centralizado su uso en la clase **NFCController** y el procesamiento de mensajes en el método **onNewIntent** de la **MainActivity**.

ANDROID BEACON LIBRARY

FUNCIONALIDAD

Android Beacon Library surge de un proyecto llamado **AltBeacon,** cuyo objetivo es crear una especificación abierta de cómo deberían estructurarse los mensajes transmitidos por dispositivos **Bluetooth Low Energy (BLE),** ya que no existe ninguna especificación para homogenizar este tipo de mensajes, y esto se estaba convirtiendo en un gran problema, ya que cada fabricante podría implementar el suyo propio [55, 56].

AltBeacon pone a nuestra disposición una librería **Open Source (Android Beacon Library)** cuyo objetivo es el de todas las librerías: facilitar el uso de un recurso. En este caso facilita el uso de los dispositivos **Beacons** para programadores **Android**. Como es lógico, esta librería es totalmente compatible con dispositivos que sigan la especificación que ellos han definido para la transmisión de mensajes BLE, pero también son compatibles con dispositivos que implementen otras especificaciones, haciendo pequeñas adaptaciones (especificar cómo se estructuran los mensajes mediante un parser llamado **BeaconParser**).

EJEMPLO DE USO

Lo primero que tendremos que hacer para poder utilizar esta librería será añadirla en nuestro proyecto. Para ello tenemos que modificar el fichero **build.gradle** donde se definen las dependencias de la aplicación. En este fichero tendremos como repositorio **jcenter()**, si no lo tenemos ya, y en el apartado de dependencias añadir: **compile 'org.altbeacon:android-beacon-library:2+'.** Si ponemos **@aar** después de la versión entonces nos descargara la versión binaria de la librería, es decir si escribimos lo siguiente: **compile 'org.altbeacon:android-beacon-library:2+@aar'.** Con estos dos pasos anteriores Gradle se encargará de descargar la librería del repositorio **jcenter**.

Lo siguiente que tenemos que hacer es añadir los permisos necesarios para poder acceder a los recursos Bluetooth en el fichero **AndroidManifest.xml:**

```
<!-- Para que la libreria AltBeacon pueda obtener la configuracion del dispositivo para calcular la distancia -->
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<!-- Para escanear Beacons -->
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

Ilustración 64 - Permisos para Android Beacon Library

Otra cosa que es importante hacer, aunque no obligatoria es añadir como requisito hardware necesario para esta aplicación que el dispositivo tenga Bluetooth. Esto se hace también en el fichero **AndroidManifest.xml** mediante el uso de **<uses-feature>:**

```
<uses-feature android:name="android.hardware.wifi"/>
<uses-feature android:name="android.hardware.bluetooth"/>
<uses-feature android:name="android.hardware.nfc"/>
```

Ilustración 65 - Requisitos Hardware

Con esto nos aseguraremos que nuestra aplicación sólo se puede instalar en dispositivos que cumplan estas características.

Llegados a este punto, ya podemos hacer uso de la librería. Esta librería tiene un setup que debe ser como el siguiente:

```
@Override
public void onCreate() {
    Log.i(TAG, "En onCreate");
    super.onCreate();
    isRunning = true;
    beaconManager = BeaconManager.getInstanceForApplication(this);
    // Layout para detectar beacons que siguen el estandar IBeacons
    beaconManager.getBeaconParsers().add(new BeaconParser().
            setBeaconLayout("m:2-3=0215,i:4-19,i:20-21,i:22-23,p:24-24"));
    // Otro layout para detectar otros dispositivos beacons
    beaconManager.getBeaconParsers().add(new BeaconParser().
            setBeaconLayout("m:2-3=0203,i:5-5,i:7-8l,i:14-19l,d:10-13,p:9-9"));
    beaconManager.setBackgroundMode(false);
}
```

Ilustración 66 - Set up de Android Beacon Library I

En el método **onCreate** del servicio que he creado para gestionar los Beacons (**BeaconService**) lo primero que hago es obtener la instancia del **BeaconManager** y añadir una serie de **BeaconParser** para que detecte el **Beacon** que estoy utilizando, ya que no sigue el estándar **AltBeacon.** Como se puede ver en la imagen anterior, también pongo el modo *background* a false, aunque no sería necesario ya que por defecto está desactivado.

Posteriormente, en el método **onStartCommand**, lo primero que hago es comprobar que el dispositivo es compatible con **Bluetooth LE.** Esto en teoría no sería necesario ya que al hacer uso de esta librería se añade como requisito para la instalación que sea compatible con **Bluetooth LE**, y por tanto, no sería necesario comprobarlo, pero de esta forma nos aseguramos de que se cumpla.

También es necesario que esté el Bluetooth encendido pero, en este caso, como es un servicio y desde un servicio no se puede interactuar con el usuario para activarlo cuando no lo esté, he decidido que esto es una precondición para usar el servicio, y por lo tanto, hay que asegurarse que está encendido antes de arrancar el servicio (esto se hace en el **BeaconManagerFragment** en el método **onResume**).

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Log.i(TAG, "En onStartCommand");
    if (checkBluetoothLESupport()) {
        startScan();
    }

    return START_STICKY;
}
```

Ilustración 67 - Setup de Android Beacon Library II

Posteriormente, en el método **startScan** comenzaremos el escaneo en busca de Beacons. Lo que haremos en este método es primero comprobar que no estamos ya enlazados y si es así llamamos al método **bind** del **BeaconManager** para enlazar el servicio.

```
private void startScan() {
    Log.i(TAG, "En startScan Bound: " + beaconManager.isBound(this));

    if (!beaconManager.isBound(this)) {
        Log.d(TAG, "Binding");
        beaconManager.bind(this);
    }
}
```

Ilustración 68 - Inicio de escaneo Beacon

Cuando se realice el enlace, se llamará a un método que se establece en la interfaz **BeaconConsumer** llamado **onBeaconServiceConnect**. En este método lo que vamos a hacer es establecer los tipos de eventos a los que nos queremos subscribir. Para ello primero tenemos que haber definido la región, que lo que hace es filtrar los tipos de Beacons que nos interesa. En mi caso la región que he definido filtrará los Beacons que tengan un determinado **UUID**, sin importar ni el **major** ni el **minor**, para escuchar todos los **Beacons** de nuestra aplicación.

```
private String myBeaconsUUID = "12345678-90ab-cdef-1234-567800223078";
private Region myRegion = new Region("es.unex.pfg.smartfidelity", Identifier.parse(myBeaconsUUID), null, null);
```

Ilustración 69 - Región Beacon

Luego establecemos los eventos a los que nos queremos subscribir, en este caso nos vamos a subscribir a eventos de **Monitoring** y **Ranging** (los dos tipos que hay), de la siguiente forma:

```
@Override
public void onBeaconServiceConnect() {
    Log.i(TAG, "En onBeaconServiceConnect: Estableciendo monitores. isBound=" + beaconManager.isBound(this));

    // Cuando se ejecuta este metodo isBound siempre es TRUE porque se ejecuta cuando el bind se ha realizado
    // Desactivo el BackgroundMode
    Log.i(TAG, "BackgroundMode Desactivado");
    beaconManager.setBackgroundMode(false);

    setMonitorNotifier();
    try {
        beaconManager.startMonitoringBeaconsInRegion(myRegion);
    } catch (RemoteException e) {
        Log.e(TAG, "Can't monitoring beacons", e);
    }

    setRangeNotifier();
    try {
        beaconManager.startRangingBeaconsInRegion(myRegion);
    } catch (RemoteException e) {
        Log.e(TAG, "Can't range beacons", e);
    }
}
```

Ilustración 70 - Subscripción a eventos de la Región

El evento **Monitoring** nos informa de cada cambio en el estado con respecto a una región (entramos o salimos de ella). En la siguiente imagen podemos ver cómo se subscribe a este tipo de eventos:

```
private void setMonitorNotifier() {
    Log.i(TAG, "En setMonitorNotifier");

    // Monitoring
    beaconManager.setMonitorNotifier(new MonitorNotifier() {
        @Override
        public void didEnterRegion(Region region) {
            Log.i(TAG, "I just saw an beacon for the first time!");
        }

        @Override
        public void didExitRegion(Region region) { Log.i(TAG, "I no longer see an beacon"); }

        @Override
        public void didDetermineStateForRegion(int state, Region region) {
            Log.i(TAG, "I have just switched from seeing/not seeing beacons: " + state);
        }
    });
}
```

Ilustración 71 - Ejemplo suscripción a Monitoring

El evento **Ranging** lo que hace es ejecutar un método que debemos implementar cada vez que cambie la distancia con el **Beacon** detectado, y nos permitirá calcular cuál es esa distancia.

```
private void setRangeNotifier() {
    Log.i(TAG, "En setRangeNotifier");
    // Ranging
    beaconManager.setRangeNotifier(new RangeNotifier() {
        @Override
        public void didRangeBeaconsInRegion(Collection<Beacon> beacons, Region region) {
            int beaconDistance;
            String minor, major;
            Beacon beacon;
            Double distancia;
            if (beacons.size() > 0) {
                beacon = beacons.iterator().next();
                distancia = beacon.getDistance();
                Log.i(TAG, "The first beacon I see is about " + myNumberDecimalFormat.format(distancia) + " meters away.");
                if (distancia < 3) {
                    beaconDistance = BEACON_NEAR;
                } else {
                    beaconDistance = BEACON_FAR;
                }

                major = String.format(formatNumber, beacon.getId2().toInt());
                minor = String.format(formatNumber, beacon.getId3().toInt());
```

Ilustración 72 - Ejemplo suscripción a Ranging

Otro dato importante es que, como ya he comentado antes, esta librería nos permite ahorrar batería en nuestro dispositivo mediante la activación del **background mode**. En este caso, al ser un servicio, no conocemos cuando la aplicación está visible o no y, por lo tanto, no podemos activar o desactivar este modo desde el servicio, sino que tienen que ser las actividades o, como en este caso, el fragmento que lo utiliza (**BeaconManagerFragment**) el que active/desactive este modo.

```
@Override
public void onPause() {
    Log.i(TAG, "En onPause");
    if (BeaconService.isRunning) {
        BeaconManager.getInstanceForApplication(mActivity).setBackgroundMode(true);
        Log.i(TAG, "BackgroundMode Activado");
    }

    super.onPause();
}
```

Ilustración 73 - Activación del background mode en BeaconManagerFragment

En la imagen anterior vemos cómo en el método **onPause** del fragmento **BeaconManagerFragment** se activa el modo de ahorro de energía. La desactivación de este modo se realiza en el método **onResume** que llamará al método **startService** y en este método, como el servicio ya está arrancado, se desactivará este modo de ahorro de energía.

CONCLUSIÓN

Esta librería ha sido un poco difícil conseguir hacerla funcionar correctamente y realizar un servicio que funcione para detectar **Beacons**, ya que hay que tener bien claro cómo funciona la librería. Por ejemplo cuando yo comencé a trabajar con ella no estaba bien documentado el uso de los **BeaconParsers** para poder trabajar con dispositivos que no sigan el estándar establecido por **AltBeacons**, sin embargo, actualmente en la web oficial ya aparece un mensaje de alerta para tener en consideración este aspecto:

Monitoring Example Code

```
public class MonitoringActivity extends Activity implements BeaconConsumer {
    protected static final String TAG = "MonitoringActivity";
    private BeaconManager beaconManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_ranging);
                beaconManager = BeaconManager.getInstanceForApplication(this);
        // To detect proprietary beacons, you must add a line like below corresponding to your beacon
        // type.  Do a web search for "setBeaconLayout" to get the proper expression.
        // 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"));
        beaconManager.bind(this);
    }
```

Otro aspecto que tampoco aparece en la documentación oficial, y que no es tan trivial, es cómo **controlar que el dispositivo tenga el Bluetooth activado** y, si no lo está, activarlo. En los ejemplos que tienen en la web (como el de la imagen anterior) sólo explican cómo funciona la librería suponiendo que está el **Bluetooth** encendido, lo que puede suponer otro impedimento.

Una vez conseguido hacerla funcionar, esta librería ha sido de gran ayuda para trabajar con **Beacons** y recomendaría su uso a cualquier persona que esté interesada en trabajar con este tipo de dispositivos. Si no tuviese esta librería y tuviese que detectar un dispositivo **Beacon** o calcular la distancia a la que se encuentra el **Beacon** de forma manual, sería una tarea muy difícil de realizar y no me hubiese permitido avanzar en otros temas.

FIREBASE

FUNCIONALIDAD

Firebase nos ofrece una API para el almacenamiento de los datos de la aplicación. Como ya se ha hablado antes en este documento, Firebase ofrece muchas características que van más allá del simple almacenamiento de datos, como pueden ser: **Multiplataforma**, **BD en tiempo real, Caché, Modo offline, Autenticación, Seguridad, Panel de administración…**

EJEMPLO DE USO

Como en las anteriores librerías, **Firebase** también necesita que la aplicación disponga de permisos, aunque en este caso sólo necesita el permiso de acceso a internet:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Ilustración 74 - Permiso de acceso a internet

Al ser una librería externa, es necesario añadirla como dependencia al fichero **build.gradle**, y además hay que poner unas opciones extra para que funcione correctamente. Primero dentro del apartado **Android** hay que poner las siguientes opciones:

```
//    Estas opciones son necesarias para evitar problemas al utilizar la libreria de Firebase
    packagingOptions {
        exclude 'META-INF/LICENSE'
        exclude 'META-INF/LICENSE-FIREBASE.txt'
        exclude 'META-INF/NOTICE'
    }
```

En el apartado **dependencies** añadiremos la dependencia de la librería.

```
compile 'com.firebase:firebase-client-android:2.3.+'
```

Otra cosa importante que hay que hacer, es comprobar cada vez que arranca la aplicación que está activada la persistencia de datos frente a reinicios, es decir, que la caché de los datos no se borrará al reiniciar el teléfono (aunque es opcional). Esto se debería hacer en el método **onCreate** de la **MainActivity**.

```
private void setUpFirebase() {
    Firebase.setAndroidContext(this);
    /**
     * Habilito la persistencia al reinicio en caso de que no este activado.
     * Solo se puede activar una vez por app, por eso hay que comprobar
     * que no este ya habilitado
     * */
    if (!Firebase.getDefaultConfig().isPersistenceEnabled())
        Firebase.getDefaultConfig().setPersistenceEnabled(true);
}
```

Ilustración 75 - Comprobación de persistencia de Firebase

Para acceder a los datos de la aplicación y poder almacenarlos en la nube, Firebase nos proporciona una URL en su servidor, en el cual expone una API REST para poder acceder a los datos, y también se utiliza esta URL para el acceso desde la API de Android. Para mantener esta URL en un único lugar y facilitar su mantenimiento, se ha creado una constante para esta URL y también para la versión de la BD actual:

```
public class ConstantsUtil {
    public static final String FIREBASE_URI_BASE = "https://luminous-heat-814.firebaseio.com";
    public static final String FIREBASE_URI_DATA = FIREBASE_URI_BASE + "/Prueba3";
    public static final int USER_NOT_LOGGED = 0;
    public static final int CUSTOMER_USER_LOGGED = 1;
    public static final int ADMIN_USER_LOGGED = 2;
}
```

Ilustración 76 - Clase ConstantsUtil

Uno de los apartados importantes de esta API es la **Autenticación de usuarios**. Para habilitar la autenticación de usuarios sólo hay que ir al **panel de administración** y elegir qué **sistemas de autenticación** queremos habilitar, y posteriormente, en la aplicación, añadir un *listener* para escuchar **eventos de autenticación**.

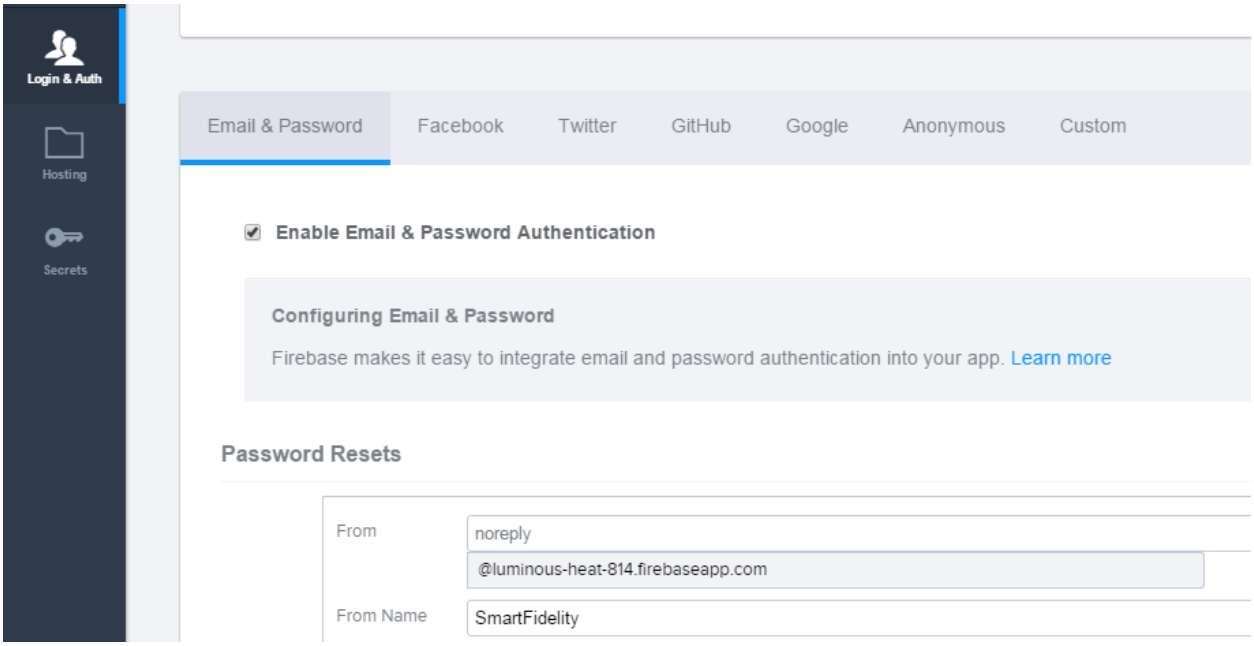


Ilustración 77 - Sistemas de autenticación de Firebase

```
private void setUpFirebaseStatusLoginListener() {
    final Firebase mFirebaseRef = new Firebase(ConstantsUtil.FIREBASE_URI_DATA);
    Firebase mFirebaseRefBase = new Firebase(ConstantsUtil.FIREBASE_URI_BASE);
    mFirebaseRefBase.addAuthStateListener((authData) -> {
        if (authData != null) {
            // user is logged in
//          Toast.makeText(getApplicationContext(), "Nuevo usuario Logueado", Toast.LENGTH_SHORT).show();

            mFirebaseRef.child("Usuarios").child(authData.getUid()).addListenerForSingleValueEvent(new ValueEventListener() {
                @Override
                public void onDataChange(DataSnapshot dataSnapshot) {
                    Usuario usuario = dataSnapshot.getValue(Usuario.class);
                    final String newUserId = dataSnapshot.getKey();
                    Boolean needUpdateCurrentShop = false;
                    setUsuario(usuario);
```

Ilustración 78 - Listener para eventos de autenticación de Firebase

En el panel de administración, también podremos configurar el mail para **restablecimiento de contraseñas** y **ver los usuarios que están actualmente registrados**.

Registered Users refresh list

Email	Created	User UID
a@a.com	Nov 01, 2015	07174022-2979-49df-8ece-e5f5d6f637ed
b@b.com	Nov 01, 2015	ae13fe63-8706-4c5f-b9a8-8d5f4f2df669

Ilustración 79 - Usuarios registrados en Firebase

Para registrar un nuevo usuario desde la aplicación en Firebase, lo que tendremos que hacer es crear un formulario en Android, y posteriormente, una vez validado que los campos son correctos, llamar a Firebase al método **createUser** pasándole el usuario y la contraseña (en este sistema de autenticación, en otros será distinto):

```
if (isValidForm()) {
    String email = inputEmail.getText().toString();
    String password = inputPassword.getText().toString();
    Log.d(TAG, "The form is valid. Email: " + email + " Password: " + password);
    mFirebaseRefBase.createUser(email, password, new Firebase.ValueResultHandler<Map<String, Object>>() {
        @Override
        public void onSuccess(Map<String, Object> result) {
            String uid = (String) result.get("uid");
            Snackbar.make(getView(), "Usuario creado correctamente", Snackbar.LENGTH_SHORT).show();
```

Ilustración 80 - Ejemplo de registro de usuario en Firebase

Para realizar consultas a la base de datos, **Firebase** ofrece dos alternativas principales: *listener* para un único evento (consulta) o *listener* para cambios en tiempo real. En este caso se ha utilizado principalmente el primero, ya que sólo se hacen consultas normales a la base de datos, pero se podrían establecer *listeners* para actualizar y recibir notificaciones en la aplicación cada vez que se produzca un determinado cambio en la base de datos. La forma de establecer un *listener* para una consulta normal es:

```
getMyFirebaseRef().child("Tiendas").addListenerForSingleValueEvent(new ValueEventListener() {

    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        Log.d(TAG, "En onDataChange de updateList");

        // Procesamos el resultado
        Iterator<DataSnapshot> iterator = dataSnapshot.getChildren().iterator();
        Tienda tiendaRead;
        DataSnapshot tiendaDataSnapshot;
        List mList = getmList();
        mList.clear();
        byte[] bitmapdata = Utils.getByteArrayOfDrawable(getResources().getDrawable(R.drawable.icon, null), Bitmap.CompressFormat.PNG);
        while (iterator.hasNext()) {
            tiendaDataSnapshot = iterator.next();
            tiendaRead = tiendaDataSnapshot.getValue(Tienda.class);
            // Establezco el minor, para ello utilizo la clave
            tiendaRead.setId(tiendaDataSnapshot.getKey());
            // TODO Change when the logo is stored in the Firebase
            tiendaRead.setLogo(bitmapdata);
            mList.add(tiendaRead);
        }

        updateRecyclerView();
    }
```

Ilustración 81 - Consulta de todas las tiendas que hay en la aplicación

El método **getMyFirebaseRef** construye una instancia de Firebase que conecta con la base de datos como se vio en la imagen anterior.

Firebase también permite realizar consultas más complejas, por ejemplo filtrando por el valor de un campo. En el siguiente ejemplo podemos ver cómo obtenemos el identificador de la tienda asociado con un usuario administrador que se acaba de *loguear* en la aplicación.

```
mFirebaseRef.child("Tiendas").orderByChild("propietario").equalTo(dataSnapshot.getKey()).addChildEventListener(new ChildEventListener() {
    @Override
    public void onChildAdded(DataSnapshot dataSnapshot, String s) {
        String tiendaId = dataSnapshot.getKey();
        SharedPreferencesManager.setCurrentShop(tiendaId, getApplicationContext());
        Log.d(TAG, "Actualizada la tienda actual a: " + tiendaId);
        continueLogin(newStatusLoginFinal, newUserId);
    }
```

Ilustración 82 - Ejemplo filtrado en consultas Firebase

La imagen anterior necesita un poco de explicación, por un lado con la referencia de Firebase consultamos las tiendas, las ordenamos por propietario (que es un campo de una tienda) y luego nos quedamos con el que en ese campo propietario tenga el valor dataSnapshot.getKey(), que es el *UID* del usuario logueado en una consulta previa. DataSnapshot es un objeto que utiliza Firebase para devolver las respuestas en formato JSON y que mediante el método **getValue** al que le pasamos la clase con la que tiene que mapear el valor, nos genera un objeto de la clase que le hemos pasado, como se puede ver en la imagen "consulta de todas las tiendas que hay en la aplicación".

Para almacenar datos en Firebase se hace accediendo a la ruta a la que queremos escribir mediante el método **child** (al igual que en las consultas) y cuando estamos en la ruta en la que queremos escribir simplemente llamamos al método **setValue**.

```
// Guardo los puntos globales
Log.d(TAG, "Guardando nuevo valor de puntos globales: " + totalPuntosGlobales);
myFirebaseRef.child("Usuarios").child(uid).child("puntos").setValue(totalPuntosGlobales);

// Guardo los puntos locales
Log.d(TAG, "Guardando nuevo valor de puntos en tienda: " + puntosEnTienda);
myFirebaseRef.child("Tiendas").child(tiendaID).child("puntos").child(uid).setValue(puntosEnTienda);
```

Ilustración 83 - Ejemplo de acumulado de puntos

Si queremos guardar un objeto completo, se haría de la misma forma:

```
Usuario usuario = new Usuario(inputName.getText().toString(), isPropietario.isChecked(), 0, inputEmail.getText().toString());
mFirebaseRef.child("Usuarios").child(uid).setValue(usuario);
```

Ilustración 84 - Ejemplo inserción de usuario en Firebase

CONCLUSIÓN

Firebase ha sido un gran descubrimiento, ya que la API que ofrece para todos los sistemas es muy sencilla de utilizar y ofrece muchas ventajas y facilidades. Si no se hubiera utilizado **Firebase** en este proyecto sino otro sistema, quizás el proyecto no hubiera ofrecido las características que ofrece.

Lo único malo que tiene **Firebase** es su curva de aprendizaje. Además, debe valorarse muy bien cómo se van a almacenar los datos en esta BD, ya que no es trivial. Pero la parte buena es que está todo muy bien documentado y dedicándole un poco de tiempo a leer la documentación, al final consigues entender fácilmente cómo empezar a trabajar con ella.

MATERIALDRAWER

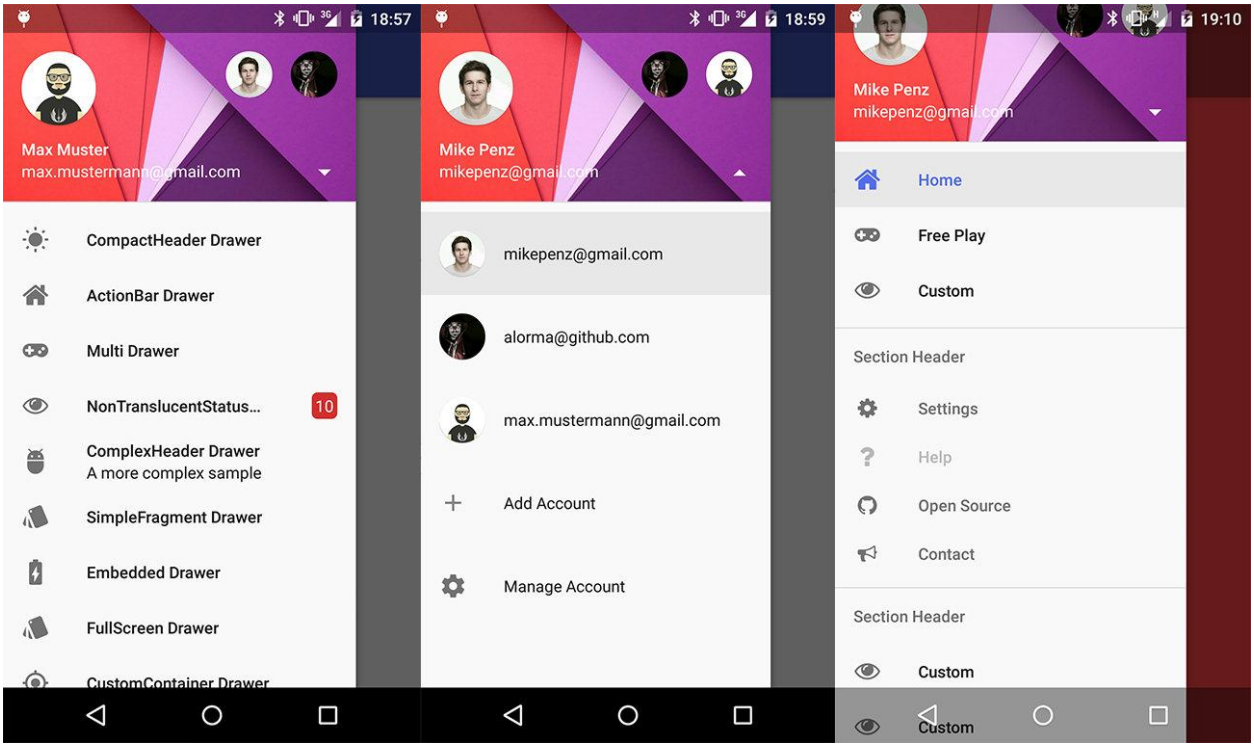


Ilustración 85 - Ejemplo MaterialDrawer

FUNCIONALIDAD

MaterialDrawer es una librería cuyo propósito es facilitar la utilización del **Navigation Drawer de Material Design [58]**. Cuando se empezó a desarrollar la aplicación, Android acababa de sacar su propia librería para la implementación del **Navigation Drawer** y ésta fue la primera que se intentó utilizar para este proyecto, pero al estar tan reciente tenía algunos fallos y era un poco difícil de utilizar, por lo que se optó por mirar otras alternativas que estuviesen más pulidas.

Esta librería permite, de forma sencilla, crear una **Navigation Drawer** a nuestro gusto y tiene gran cantidad de opciones. Entre sus cualidades se encuentran:

- Soporte a Multiusuario con selector de usuario
- Opciones separadas por secciones
- Iconos en las opciones del menú
- Soporte desde API 10 de Android
- Soporte de SVG para los iconos
- Varios temas
- Basado en RecyclerView
- Buena documentación

EJEMPLO DE USO

Lo primero que hay que hacer para utilizar esta librería es añadir la dependencia en el fichero **build.gradle**

```
compile('com.mikepenz:materialdrawer:3.0.9@aar') {
    transitive = true
}
```

Ilustración 86 - Dependencia MaterialDrawer

La utilización de la librería es bastante sencilla. En este caso se ha creado un método llamado **setUpDrawer** en la clase **MainLibraryWithDrawer** que se encarga de construir la **Navigation Drawer** según las opciones establecidas por el fragmento concreto que extienda la clase **MainLibraryWithDrawer.**

La **Navigation Drawer** está formada por tres elementos: **el perfil del usuario, la cabecera y las opciones**. Una vez construidos estos tres elementos, se activa el **hamburger icon** en la **ActionBar** y se activa un efecto en el Drawer para cuando se abra/cierre la **Navigation Drawer.** El método **setUpDrawer** es así de sencillo:

```
protected void setUpDrawer() {
    IProfile profile = getProfileForCurrentUser();
    AccountHeader accountHeader = getAccountHeader(profile);
    mDrawer = getDrawer(accountHeader);

    enableHamburgerIconInActionBar();

    mDrawer.getActionBarDrawerToggle().setDrawerIndicatorEnabled(true);
    mDrawer.setSelection(0);
}
```

Ilustración 87 - Método setUpDrawer

```
private void enableHamburgerIconInActionBar() {
    try {
        ActionBar supportActionBar = ((AppCompatActivity) getActivity()).getSupportActionBar();
        if (supportActionBar != null) {
            supportActionBar.setDisplayHomeAsUpEnabled(false);
        }
    } catch (ClassCastException e) {
        // Nothing to do here
    }
}
```

Ilustración 88 - Habilitar el Hamburger Icon en la Action Bar

Lo primero que hacemos es construir el perfil del usuario, para ello tenemos que obtener los datos del usuario que está *logueado* actualmente. Este perfil se crea mediante una clase llamada **ProfileDrawerItem**.

```
private IProfile getProfileForCurrentUser() {
    Usuario usuario;
    try {
        usuario = ((MainActivity) getActivity()).getUsuario();
    } catch (ClassCastException e) {
        usuario = new Usuario();
    }

    String name = usuario.getTextForNavigationDrawer();

    return new ProfileDrawerItem()
            .withName(name)
            .withEmail(usuario.getEmail())
            .withIcon(getActivity().getResources().getDrawable(R.drawable.icon, null));
}
```

Ilustración 89 - Método que construye el perfil del usuario para la Navigation Drawer

```
public String getTextForNavigationDrawer() {
    if (this.getPropietario()) {
        return String.format("%s", this.getNombre());
    } else {
        return String.format("%s (%d Puntos)", this.getNombre(), this.getPuntos());
    }
}
```

Ilustración 90 - Método getTextForNavigationDrawer en la clase Usuario

Para la construcción de la cabecera del **Navigation Drawer** se utiliza la clase **AccountHeaderBuilder** que como vemos ya en su nombre, implementa el patrón de diseño **Builder**. Esta clase necesita del perfil del usuario que hemos creado en el paso anterior para construir la cabecera.

```
private AccountHeader getAccountHeader(IProfile profile) {
    return new AccountHeaderBuilder()
            .withActivity(getActivity())
            .withHeaderBackground(R.drawable.nav_header_bg)
            .addProfiles(profile)
            .build();
}
```

Ilustración 91 - Construcción de la cabecera del Navigation Drawer

Lo último necesario ya es construir la **Navigation Drawer** configurándola mediante la clase **DrawerBuilder** que también implementa el patrón de diseño **Builder**. En este caso, se decidió que las opciones que se le mostrarán al usuario en la **Navigation Drawer** sean establecidas según el tipo de usuario (Admin o Customer), por lo tanto, se utiliza un método que implementan las clases que extienden **MainLibraryWithDrawer** (**AdminLibrary y CustomerLibrary**) llamado **getDrawerItems**.

```
private Drawer getDrawer(AccountHeader accountHeader) {
    return new DrawerBuilder()
            .withActivity(getActivity())
            .withToolbar(mToolbar)
            .withActionBarDrawerToggleAnimated(true)
            .withTranslucentStatusBar(true)
            .withAccountHeader(accountHeader)
            .withOnDrawerItemClickListener(new Drawer.OnDrawerItemClickListener() {
                @Override
                public boolean onItemClick(AdapterView<?> parent, View view, int position, long id, IDrawerItem drawerItem) {
                    if (position >= 0) {
                        Log.d(TAG, "Click in position: " + position);
                        mDrawer.setSelection(position, false);
                        loadFragment();
                        mDrawer.closeDrawer();
                    }
                    return true;
                }
            })
            .addDrawerItems(getDrawerItems()) // Add the options for drawer
            .build();
}
```

Ilustración 92 - Construcción de la Navigation Drawer

```
@Override
protected IDrawerItem[] getDrawerItems() {
    List<IDrawerItem> mListPrimaryDrawerItems = new ArrayList<>();
    mListPrimaryDrawerItems.add(new PrimaryDrawerItem().withName("Principal").withIcon(R.drawable.icon));
    mListPrimaryDrawerItems.add(new PrimaryDrawerItem().withName("Promociones").withIcon(R.drawable.icon));
    mListPrimaryDrawerItems.add(new DividerDrawerItem());
    mListPrimaryDrawerItems.add(new SecondaryDrawerItem().withName("Settings").withIcon(R.drawable.ic_settings));
    mListPrimaryDrawerItems.add(new SecondaryDrawerItem().withName("Logout").withIcon(R.drawable.ic_exit_to_app));

    return mListPrimaryDrawerItems.toArray(new IDrawerItem[mListPrimaryDrawerItems.size()]);
}
```

Ilustración 93 - Implementación de getDrawerItems en CustomerLibrary

CONCLUSIÓN

La elección de esta librería para la implementación de la **Navigation Drawer** creo que ha sido un acierto, ya que el código queda muy legible y la librería ofrece mucha personalización.

El resultado final [32] me parece muy bueno y creo que es mucho mejor que la opción que ofrece Google en la librería de compatibilidad.

[32] Ver resultado en el punto "MaterialDrawer" del apartado "Elementos de Material Design"

MATERIALSPINNER

FUNCIONALIDAD

MaterialSpinner es una librería cuyo objetivo es proporcionar un **Spinner** mejorado para mejorar la experiencia de usuario [59]. Este **Spinner** nos permite establecer **"etiquetas flotantes"** que siempre estará visible para el usuario, no como sucede en el **Spinner** por defecto de Android, que cuando el usuario selecciona una opción, el **hint** desaparece y se queda sin contexto para el campo.

Otra de las funcionalidades que ofrece esta librería es la de establecer **mensajes de error** en campos que no han pasado la **validación**. Esto también mejora la experiencia de usuario ya que le podemos ayudar a rellenar correctamente los datos.

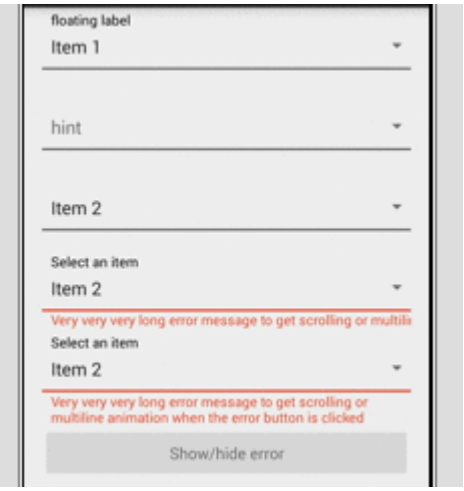


Ilustración 94 - Ejemplo MaterialSpinner

EJEMPLO DE USO

Como siempre lo primero es añadir la dependencia en el fichero **build.gradle**. En este caso vemos que tenemos que añadir una línea para que no entre en conflicto con el **appcompat** que estamos utilizando.

```
compile('com.github.ganfra:material-spinner:1.+') {
    exclude group: 'com.android.support', module: 'appcompat-v7'
}
```

Ilustración 95 - Dependencia de MaterialSpinner

Para utilizarlo, tenemos que añadirlo primero en un **layout XML**. A continuación se muestra un ejemplo uso. En este caso el diseño se lo estamos estableciendo mediante un atributo **Style**, ya que todos tendrán el mismo aspecto. Dos puntos importantes son el campo **FloatingLabelText** y el **hint.**

```
<fr.ganfra.materialspinner.MaterialSpinner
    android:id="@+id/frag_add_promocion_categoria"
    style="@style/MaterialSpinnerStyle"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignEnd="@+id/frag_add_promocion_field_titulo_layout"
    android:layout_alignStart="@+id/frag_add_promocion_field_titulo_layout"
    android:layout_below="@+id/frag_add_promocion_beacon"
    app:ms_floatingLabelText="Categoria"
    app:ms_hint="Seleccionar Categoria"
    />
```

Ilustración 96 - Uso de MaterialSpinner en AddPromotionFragment

```
<style name="MaterialSpinnerStyle">
    <item name="android:popupBackground">@color/theme_highlight</item>
    <item name="ms_alignLabels">true</item>
    <item name="ms_arrowColor">@color/theme_accent</item>
    <item name="ms_baseColor">@color/theme_accent</item>
    <item name="ms_enableErrorLabel">true</item>
    <item name="ms_enableFloatingLabel">true</item>
    <item name="ms_floatingLabelColor">@color/theme_accent</item>
    <item name="ms_highlightColor">@color/theme_highlight</item>
    <item name="ms_hintColor">@color/theme_accent</item>
    <item name="ms_multiline">false</item>
</style>
```

Ilustración 97 - MaterialSpinnerStyle

Posteriormente, podremos acceder al **MaterialSpinner** desde código para poder leer el valor seleccionado por el usuario o establecer mensajes de error en caso de que sea necesario. Es importante dejar claro, que para poder acceder a estas vistas es necesario hacerlo **como mínimo** en el método **onCreateView**, ya que antes no estarán renderizadas las vistas y dará error.

En el siguiente ejemplo podemos ver cómo se crean dos adaptadores que tendrán los datos que contendrán los **Spinners** y cómo se establecen.

```
spinnerBeaconsAdapter = new ArrayAdapter<>(
        getActivity().getApplicationContext(),
        android.R.layout.simple_spinner_item,
        mBeacons
);
spinnerBeaconsAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);

spinnerCategoriasAdapter = new ArrayAdapter<>(
        getActivity().getApplicationContext(),
        android.R.layout.simple_spinner_item,
        mCategorias
);
spinnerCategoriasAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);

spinnerBeacons = (MaterialSpinner) rootView.findViewById(R.id.frag_add_promocion_beacon);
spinnerCategorias = (MaterialSpinner) rootView.findViewById(R.id.frag_add_promocion_categoria);

spinnerBeacons.setAdapter(spinnerBeaconsAdapter);
spinnerCategorias.setAdapter(spinnerCategoriasAdapter);
```

Ilustración 98 - Ejemplo de acceso a un MaterialSpinner

Para acceder a los datos del **Spinner** o establecer un mensaje de error, se puede hacer de la siguiente forma.

```
if (validateForm()) {
    String titulo = inputTitulo.getText().toString();
    String descripcion = inputDescripcion.getText().toString();
    String beacon = mBeacons.get(spinnerBeacons.getSelectedItemPosition() - 1);
    String categoria = mCategorias.get(spinnerCategorias.getSelectedItemPosition() - 1);
```

Ilustración 99 - Acceso al valor de un MaterialSpinner

```
private Boolean validateCategoria() {
    Boolean isValid = spinnerCategorias.getSelectedItemPosition() > 0;
    if (isValid) {
        spinnerCategorias.setError(null);
    } else {
        spinnerCategorias.setError("Selecciona una categoria");
        requestFocus(spinnerCategorias);
    }

    return isValid;
}
```

Ilustración 100 - Validación de un MaterialSpinner

CONCLUSIÓN

Esta librería ha sido de gran ayuda para tener formularios mejor adaptados a los requisitos de **Material Design** y por tanto, para ofrecer una mejor experiencia de usuario.

Su uso me ha parecido bastante sencillo y el resultado final[33] es muy bueno.

TEXTINPUTLAYOUT

FUNCIONALIDAD

Este elemento forma parte de la librería de soporte de Android. Su utilidad al igual que la anterior librería, es proporcionar una mejor experiencia de usuario al rellenar formularios, ya que nos permite establecer **"etiquetas flotantes"** y establecer **mensajes de error** en los campos de texto.

[33] Ver resultado en el punto "MaterialSpinner y TextInputLayout" del apartado "Elementos de Material Design"

EJEMPLO DE USO

Lo primero que hay que hacer es añadir las dependencias necesarias, en este caso la librería **design** de la librería **Android Support**. La siguiente imagen muestra las librerías de **Android Support** utilizadas en este proyecto.

```
dependencies {
    compile 'com.android.support:appcompat-v7:22.+'
    compile 'com.android.support:support-v4:22.+'
    compile 'com.android.support:design:22.+'
    compile 'com.android.support:recyclerview-v7:22.+'
    compile 'com.android.support:cardview-v7:22.+'
```

Ilustración 101 - Dependencias de Android Support

Luego tendremos que utilizar el **TextInputLayout** recubriendo un **EditText**.

```
<android.support.design.widget.TextInputLayout
    android:id="@+id/frag_registro_usuario_field_nombre_layout"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="20dp"
    >

    <EditText
        android:id="@+id/frag_registro_usuario_field_nombre"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="12"
        android:hint="Full Name"
        android:inputType="textPersonName"
        />
</android.support.design.widget.TextInputLayout>
```

Ilustración 102 - Uso de TextInputLayout

Por último, para acceder al **TextInputLayout** desde código y poder establecer **mensajes de error** se hará de la siguiente forma. En este caso el **TextInputLayout** solamente nos proporciona un efecto visual y no contendrá ningún dato, los datos los contiene el **EditText.**

```
private boolean validateName() {
    if (inputName.getText().toString().trim().isEmpty()) {
        inputLayoutName.setError("Introduce tu nombre");
        requestFocus(inputName);
        return false;
    } else {
        inputLayoutName.setErrorEnabled(false);
    }

    return true;
}
```

Ilustración 103 - Establecer mensaje de error en TextInputLayout

CONCLUSIÓN

Este elemento de la librería de **Android Support** me ha parecido bastante interesante, ya que la experiencia que se consigue gracias a estos pequeños detalles es muy buena.

El resultado final[34] del uso de este elemento me ha parecido muy bueno y proporciona una gran ayuda al usuario.

QRDROID

FUNCIONALIDAD

QrDroid es una aplicación Android para lectura y generación de códigos QR [60, 61]. Esta aplicación proporciona un servicio de lectura y generación de códigos QR para otras aplicaciones y en este proyecto se ha hecho uso de este servicio.

[34] Ver resultado en el punto "MaterialSpinner y TextInputLayout" del apartado "Elementos de Material Design"

EJEMPLO DE USO

Antes de empezar, es necesario que esté instalada la aplicación **QrDroid** en el teléfono, ya que se utilizara para delegar la lectura y generación de códigos QR. En las siguientes imágenes se puede ver cómo se hace la lectura y generación de un código QR utilizando el servicio que nos proporciona la aplicación **QrDroid**.

```
Button buttonScanQR = (Button) mActivity.findViewById(R.id.frag_registro_compra_button_scan_qr);
buttonScanQR.setOnClickListener((v) -> {
        Intent qrDroid = new Intent(QrDroidConstants.SCAN);
        qrDroid.putExtra(COMPLETE,true);    // Opcional - Para que devuelva el resultado completo. Ej. tel:+123456
        try {
            startActivityForResult(qrDroid, QrDroidConstants.REQUEST_SCAN_QR);
        } catch (ActivityNotFoundException e) {
            QrDroidConstants.askUserToGetQRDroid(mActivity);
        }
});
```

Ilustración 104 - Ejemplo de lectura de código QR

```
buttonGenQR.setOnClickListener((v) -> {
        Intent qrDroid = new Intent(QrDroidConstants.ENCODE);
        String uid = SharedPreferencesManager.getUidUserLogged(getActivity().getApplicationContext());
        if (uid.trim().length() == 0) {
            Snackbar.make(getView(), "Texto Vacio", Snackbar.LENGTH_LONG).show();
            return;
        }
        qrDroid.putExtra(QrDroidConstants.CODE, uid);
        //Optionally, set requested image size. 0 means "Fit Screen"
        qrDroid.putExtra(QrDroidConstants.SIZE, 0);
        qrDroid.putExtra(QrDroidConstants.IMAGE, true);    // Indicamos que queremos que guarde la imagen en la SD
        try {
            startActivityForResult(qrDroid, QrDroidConstants.REQUEST_ENCONDE_QR);
        } catch (ActivityNotFoundException e) {
            QrDroidConstants.askUserToGetQRDroid(mActivity);
        }
});
```

Ilustración 105 - Generación de código QR

Como se puede ver en ambos casos tenemos que poner un *try catch* ya que puede que la aplicación no esté instalada. En caso de que la aplicación no esté instalada, le solicitaremos al usuario que la instale para poder utilizar las funcionalidades de los códigos QR.

```
public static void askUserToGetQRDroid(final Activity mActivity) {
    Toast.makeText(mActivity.getApplicationContext(), "QR Droid Not Found", Toast.LENGTH_SHORT).show();

    //Apparently, QR Droid is not installed, or it's previous to version 3.5
    AlertDialog.Builder builder = new AlertDialog.Builder(mActivity);
    builder.setMessage("QR Droid version 3.5 or later is required. Do you want ...")
            .setCancelable(true)
            .setNegativeButton("Cancel", (dialog, id) -> {
                dialog.cancel();
            })
            .setPositiveButton("Download", (dialog, id) -> {
                mActivity.startActivity(new Intent(
                        Intent.ACTION_VIEW,
                        Uri.parse("http://market.android.com/details?id=la.droid.qr")
                ));
            });
    builder.create().show();
}
```

Ilustración 106 - Implementación del método askUserToGetQRDroid

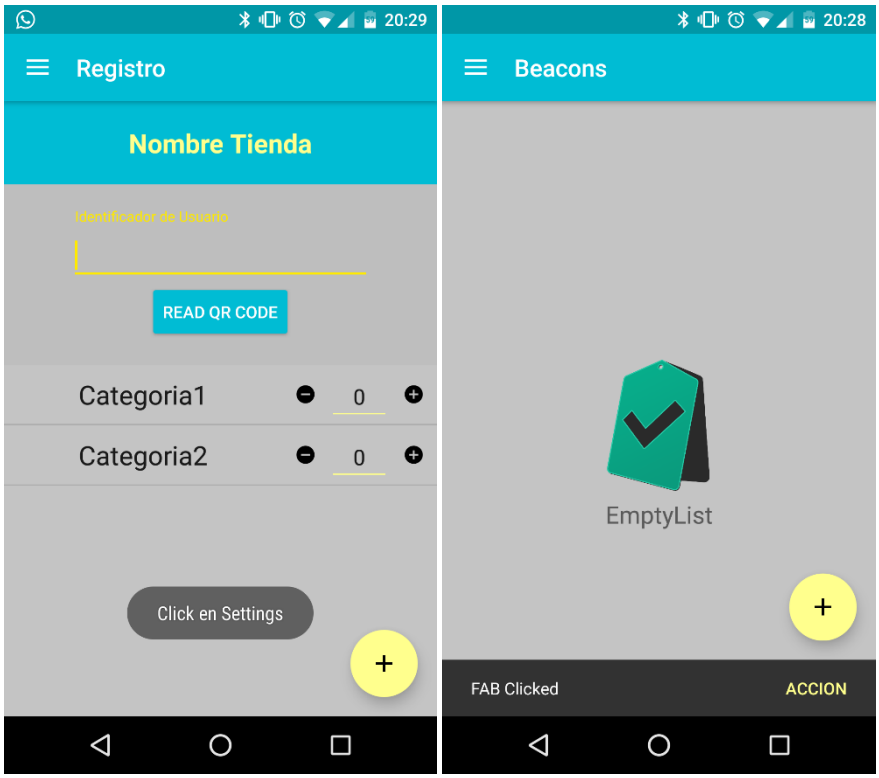
CONCLUSIÓN

Esta solución para añadir soporte de códigos QR a la aplicación me ha parecido extremadamente sencilla y con unos resultados inmejorables, ya que los resultados son muy buenos. Si tuviese que hacer otra aplicación con soporte para códigos QR utilizaría el servicio de **QrDroid** sin dudarlo.

5.3.4. ELEMENTOS DE MATERIAL DESIGN

SNACKBAR

La **SnackBar** es un nuevo elemento añadido en **Material Design** para proporcionar feedback al usuario sobre una operación que ha realizado, anteriormente se utilizaban los **Toast**, e incluso permiten realizar una acción al respecto. La principal diferencia con los **Toast** es que la **SnackBar** necesita de una vista sobre la que mostrarse, mientras que los **Toast** utilizan un contexto, por ejemplo el de la aplicación, por lo que los **Toast** siguen siendo necesarios para informar al usuario en operaciones que le saquen de la aplicación, por ejemplo cuando se le manda a la configuración para activar el **NFC**. Si en ese caso se utilizase una **SnackBar**, cuando se saque al usuario de la aplicación perderá el mensaje, mientras que con el **Toast** permanece.



EMPTYRECYCLERVIEW

EmptyRecyclerView es una personalización propia de la **RecyclerView** ya que ésta no permite establecer un **EmptyCase** y con unas pequeñas modificaciones ya sí lo permite.

Estas pequeñas modificaciones son simplemente extender la clase **RecyclerView**, tener un *Observer* para comprobar si está vacía la fuente de datos cada vez que se modifique el adaptador (que nos proporciona los datos), y añadir un método que permita establecer una **View** como **EmptyCase**. Al crearse un objeto de esta clase, lo primero que se va a comprobar es si está vacía la fuente de datos y en tal caso, si hay **EmptyCase** establecida, se pondrá visible la **View** que representa el **EmptyCase**.

```
public void checkIfEmpty() {
    Log.d(TAG, "En checkIfEmpty");
    if (emptyView != null && getAdapter() != null) {
        emptyView.setVisibility(getAdapter().getItemCount() > 0 ? GONE : VISIBLE);
    }
}
```

Ilustración 107 - Implementación del método checkIfEmpty de la clase EmptyRecyclerView

La **View** que representa el **EmptyCase** deberá establecerse en el layout.

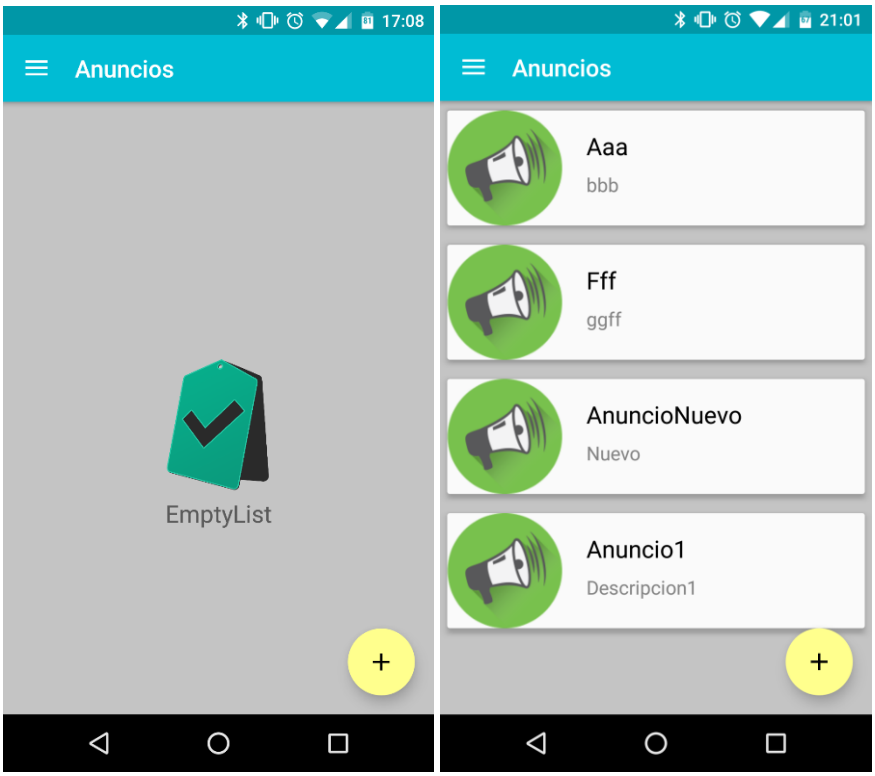
```
<!-- Vista que se utiliza como EmptyView para la EmptyRecyclerView -->
<LinearLayout
    android:id="@+id/frag_anuncios_empty_list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical">

    <ImageView
        android:layout_width="240dp"
        android:layout_height="120dp"
        android:src="@drawable/ic_empty_list"
        />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="EmptyList"
        android:textSize="20sp"
        />
</LinearLayout>
```

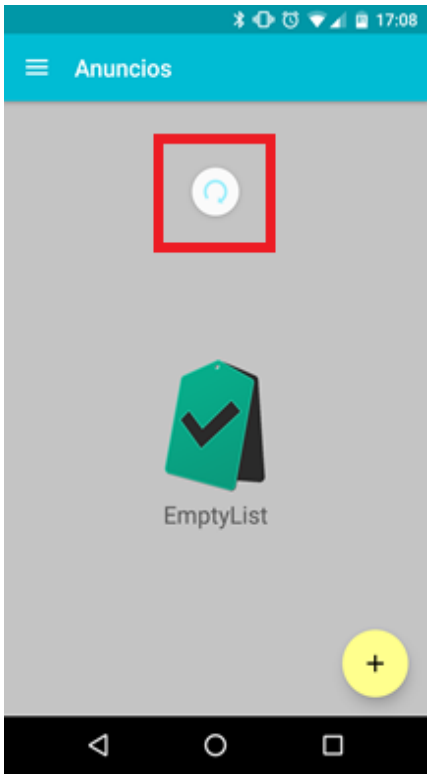
Ilustración 108 – EmptyView

El resultado final sería algo así:



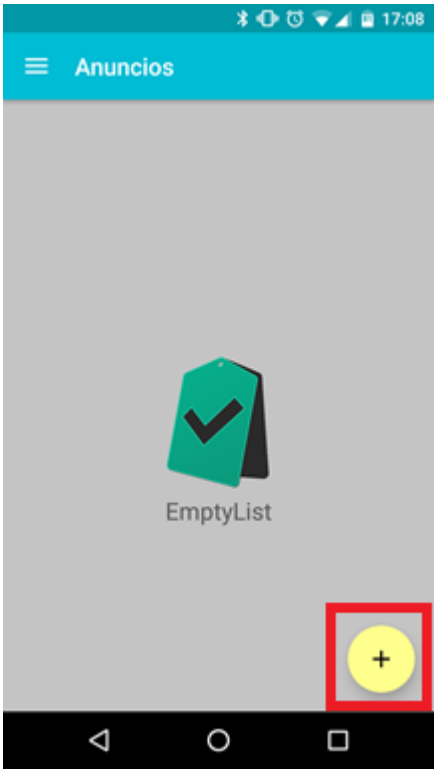
SWIPETOREFRESH

SwipeToRefresh es otra de las novedades de **Material Design**. Se trata de un nuevo patrón para actualizar la ventana actual que consiste en hacer *Swipe* desde la parte superior de la pantalla hacia abajo. Esto mostrará un pequeño círculo que irá cambiando de colores mientras se cargan los datos.



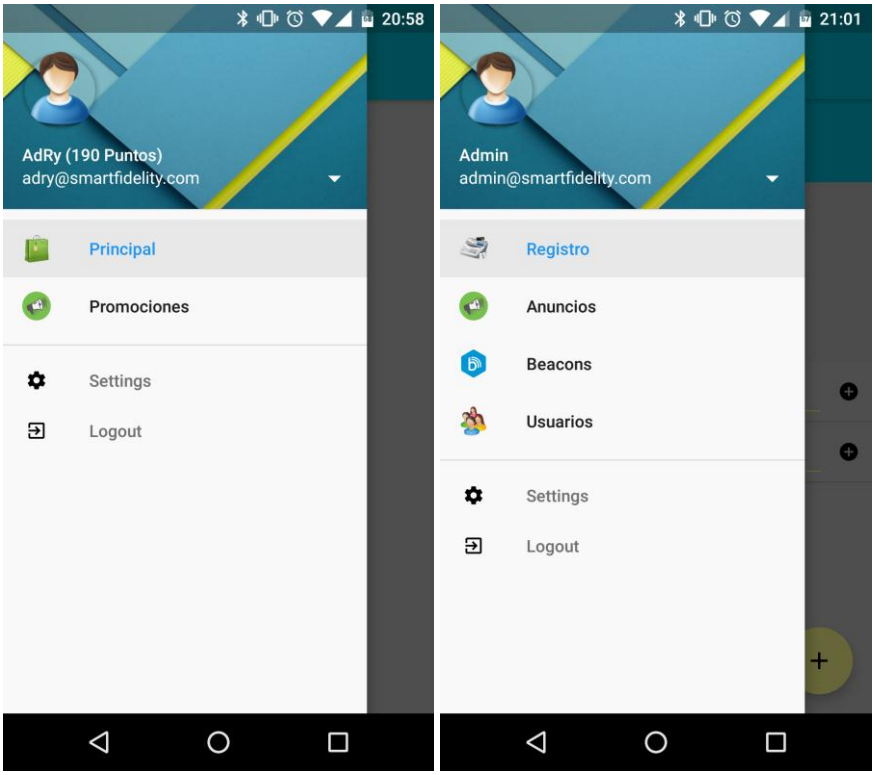
FLOATING ACTION BUTTON (FAB)

El **Floating Action Button** también conocido por sus siglas **FAB** es un **botón de acción** que se sitúa en la **parte inferior derecha de la pantalla** y que también fue introducido en **Material Design.** Este botón debe tener asignada la acción principal que puede realizar el usuario en la pantalla, por ejemplo en nuestro caso, en la pantalla de **registro de compra** la acción es registrar una compra. Sin embargo, en la lista de **Anuncios** la acción es crear un nuevo anuncio.



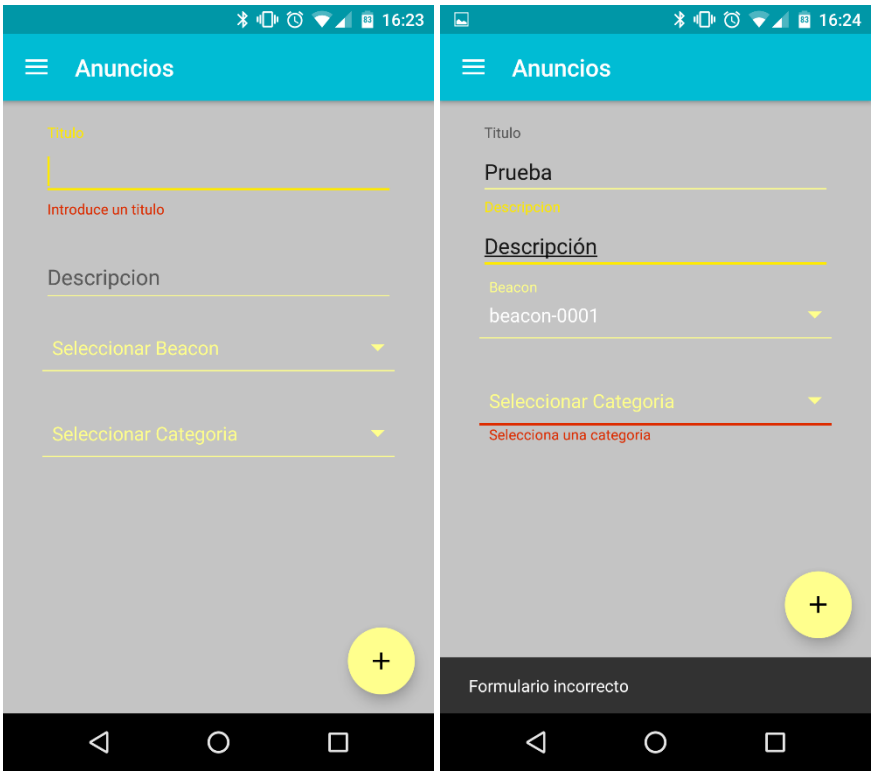
MATERIALDRAWER

Ya se ha hablado en el punto anterior de esta librería. Esta facilita la creación de una **Navigation Drawer**.



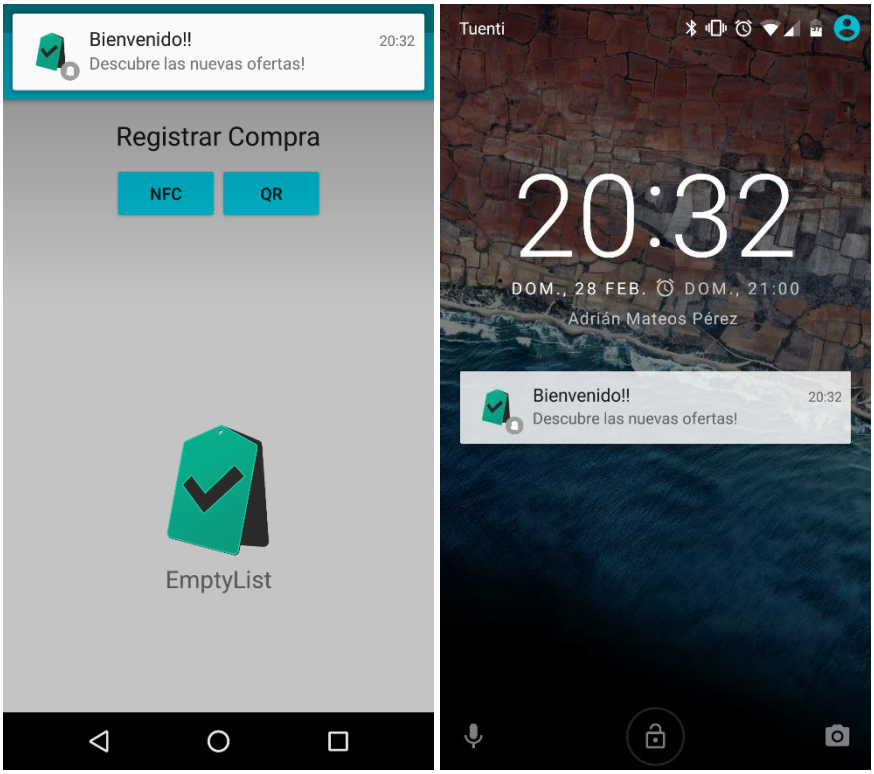
MATERIALSPINNER Y TEXTINPUTLAYOUT

Estos dos elementos se utilizan para mejorar la experiencia de usuario a la hora de rellenar formularios en la aplicación y adaptar esa experiencia lo máximo posible a **Material Design**. En la siguiente imagen podemos ver cómo funcionan estos dos elementos en el formulario que tiene un administrador de una tienda para crear anuncios.



NUEVAS NOTIFICACIONES

En **Android 6.0 Marshmallow** se ha introducido un nuevo sistema de notificaciones, que permite establecer notificaciones como prioritarias para verlas sobre otras aplicaciones, y también permite que las notificaciones se vean en la pantalla de bloqueo del dispositivo. En este proyecto, las notificaciones han sido preparadas para exprimir al máximo estas novedades, ya que es una parte fundamental de la aplicación que el usuario conozca las ofertas y novedades que tienen las tiendas.



6. MANUAL DE USUARIO

En este apartado se va a explicar qué posibilidades ofrece esta aplicación y cómo ha de utilizarse. Lo primero a tener en cuenta es que esta aplicación puede ser utilizada por **dos tipos de usuarios, clientes o propietarios,** y que cada uno tendrá distintas opciones.

6.1. LOGIN Y REGISTRO

Lo primero que verá un usuario al acceder a la aplicación será la **pantalla de *login***, donde se le solicitarán sus credenciales de acceso. En caso de que el usuario tenga estas credenciales, deberá introducirlas y podrá así acceder a la aplicación. Lo normal será que si utiliza por primera vez la aplicación no tenga estas credenciales de acceso, por lo tanto, tendrá que registrarse en la plataforma para poder acceder.

A continuación se muestra cómo vería el usuario la pantalla de *login*. Como se puede ver, sólo se le **solicita un correo electrónico y una contraseña**. También se puede ver que hay un **botón de registro** mediante el cual el usuario podrá acceder a registrarse en la plataforma.

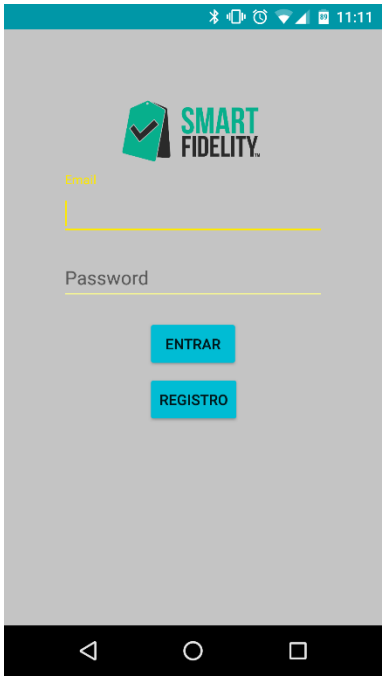


Ilustración 109 - Pantalla de Login

La siguiente imagen muestra la **pantalla de registro** de un nuevo usuario. En esta pantalla el usuario tendrá que rellenar sus datos e indicar el tipo de usuario indicando si es un cliente o un propietario (en este caso deberá marcar la opción de propietario). Para guardar el formulario, el usuario tendrá que pulsar sobre el icono de guardar (disquete), mientras que si quiere cancelar la operación, el usuario puede o bien pulsar atrás en su dispositivo o hacer clic en el icono del aspa.

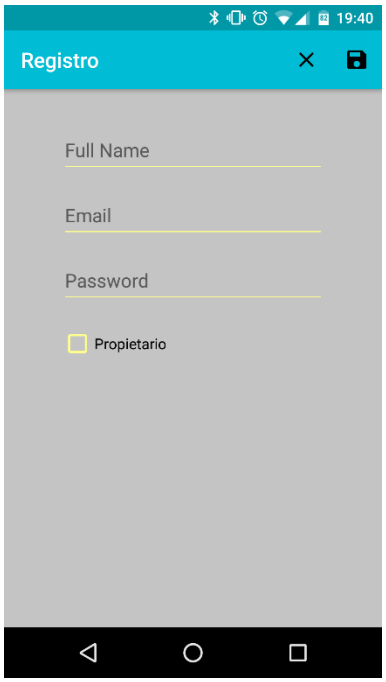


Ilustración 110 - Registro de usuarios

6.2. CASOS DE USO DEL CLIENTE

Un cliente podrá realizar las siguientes operaciones en la aplicación:

- Registrar una compra
- Consultar los puntos que tiene en tiendas en las que ha hecho compras
- Consultar las promociones activas en la plataforma

El cliente podrá acceder a estas opciones mediante la **Navigation Drawer** (una vez esté *logueado*) y en esta también podrá consultar el número de puntos globales en la plataforma. La siguiente imagen muestra un ejemplo de lo que vería un cliente en su **Navigation Drawer**.

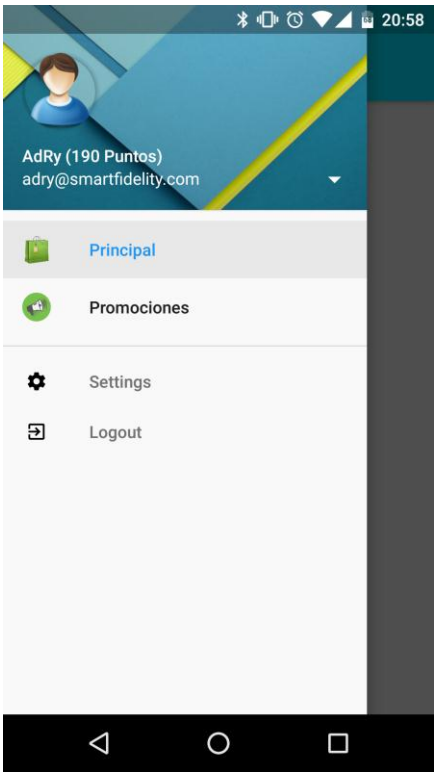


Ilustración 111 - Navigation Drawer de un clientes

La opción **Principal** sería la ventana principal que se le mostraría al usuario cada vez que accede a la aplicación. En esta ventana el usuario podrá registrar una nueva compra identificándose frente al establecimiento mediante las opciones de **NFC o Código QR,** así como consultar los puntos que tiene en las tiendas en las que ha hecho compras.

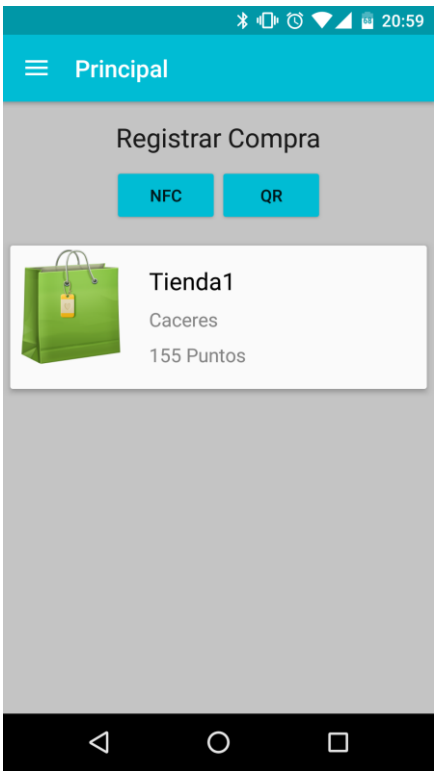


Ilustración 112 - Pantalla principal de un cliente

Para registrar una compra utilizando la opción de **NFC,** el usuario deberá pulsar el botón **NFC** y juntar su dispositivo al dispositivo del que disponga el dependiente del establecimiento. Una vez hecho esto, le saldrá un mensaje en la pantalla solicitándole que confirme el envío de datos. El usuario deberá aceptar para que se envíe su identificador de usuario.

Para registrar una compra utilizando la opción de **código QR,** el usuario simplemente tendrá que hacer clic en el botón **QR** e inmediatamente le aparecerá en su pantalla su **código QR**. Este **código QR** deberá ser leído por la aplicación del dependiente.

La opción **Promociones** mostrará al usuario una lista de promociones activas en ese momento en la plataforma.

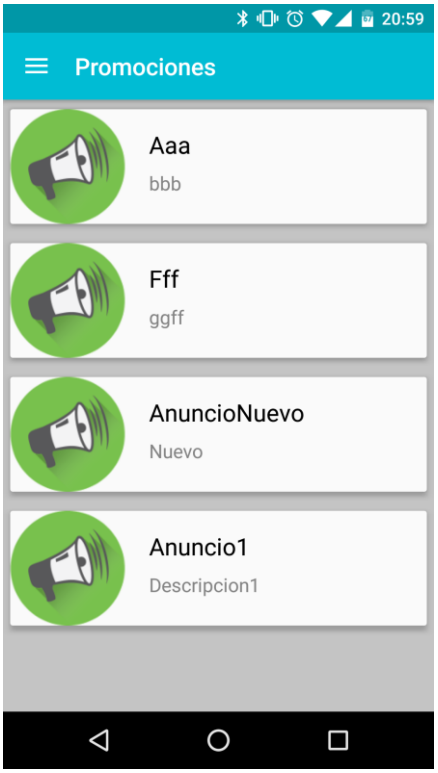


Ilustración 113 - Promociones de un cliente

6.3. CASOS DE USO DEL PROPIETARIO

Un propietario de una tienda tendrá en esta aplicación las siguientes opciones:

- Registro de compras
- Gestión de Anuncios (Promociones)
- Gestión de Beacons
- Gestión de Usuarios (Solo visualización)

El propietario tendrá a su disposición estas opciones mediante la **Navigation Drawer** (una vez esté logueado). La siguiente imagen muestra un ejemplo de lo que vería un propietario en su **Navigation Drawer**.

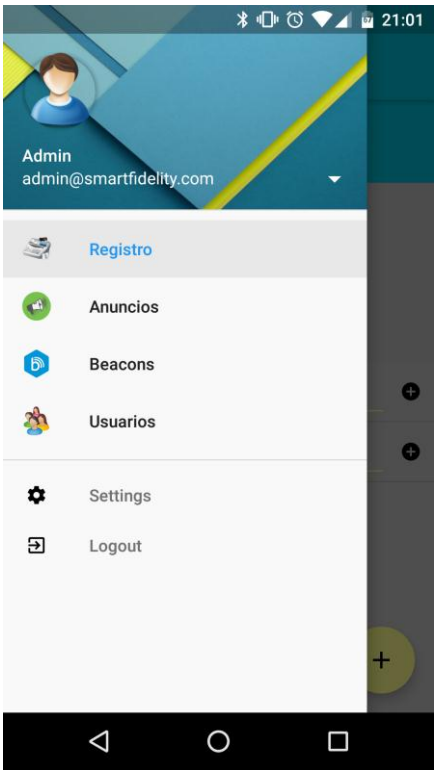


Ilustración 114 - Navigation Drawer de un propietario

La pantalla **Registro** será la pantalla que le será mostrada a un usuario que es propietario cada vez que abra la aplicación. En esta pantalla podrá registrar las compras que realicen los clientes en su establecimiento. Para **registrar una compra,** el propietario deberá introducir la **cantidad de productos de cada categoría** que ha comprado el cliente. Esto se utilizará **para calcular el número de puntos** que le serán añadidos al cliente por la compra. También deberá **identificar al cliente,** para esta tarea tiene **tres opciones:** mediante **NFC,** mediante **código QR** (puede ser generado por la aplicación del cliente o mediante una tarjeta de fidelización con el código QR) o **introduciéndolo manualmente**.

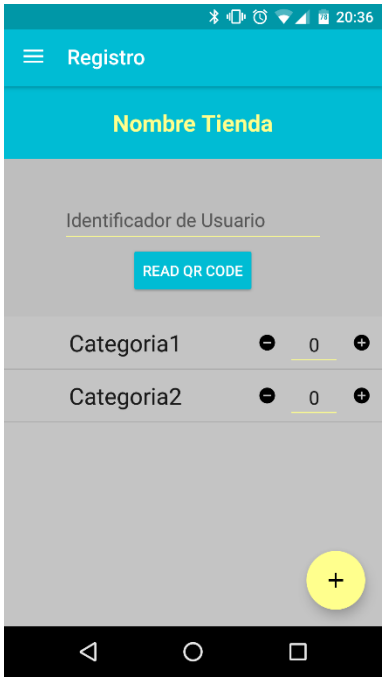


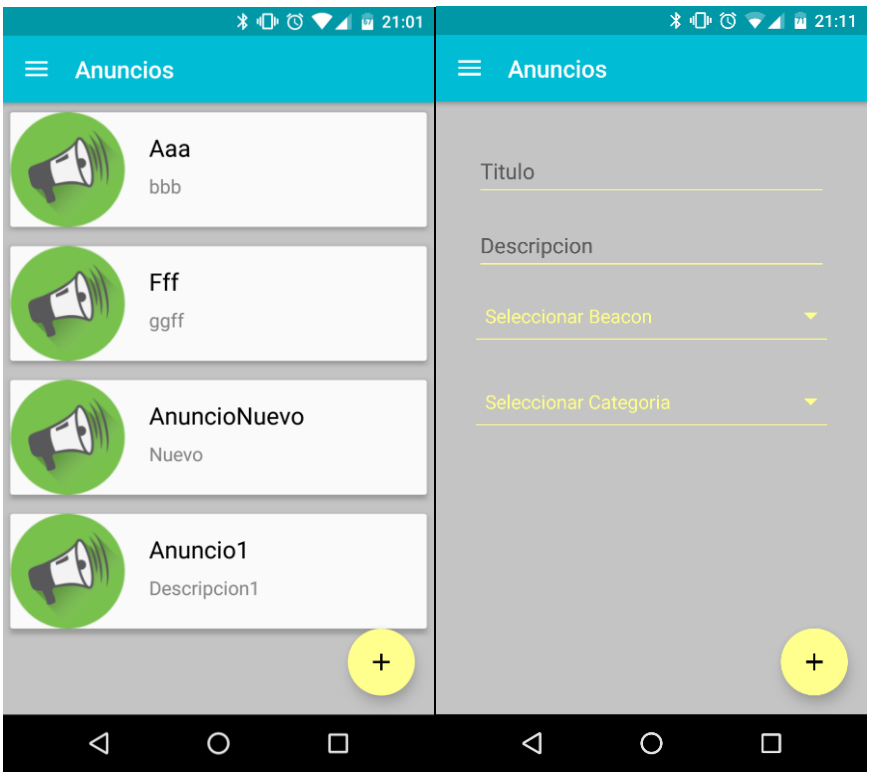
Ilustración 115 –Registro de compras para el propietario

Para **identificar al usuario mediante NFC,** el propietario lo único que tendrá que hacer es solicitar al usuario que haga clic en el botón **NFC** de su aplicación**,** juntar los dispositivos y esperar a que el usuario acepte la transferencia de datos.

Para la **identificación de usuario mediante código QR,** deberá solicitar al usuario su **código QR,** pulsar el botón **"Read QR Code"** que tiene en su pantalla y leer el **código QR** mediante la cámara.

Para la **identificación manual del usuario,** deberá solicitarle el identificador de usuario e introducirlo en el campo de texto que hay en la pantalla para tal efecto.

La pantalla **Anuncios** permite al propietario de un establecimiento consultar y añadir nuevos anuncios (promociones). Si el propietario pulsa en el botón de añadir, le aparecerá un formulario para rellenar los datos del anuncio que desea añadir, y pulsando nuevamente en el botón añadir, se añadirá el nuevo anuncio a la lista de anuncios disponibles para la tienda.



La pantalla **Beacons** muestra al propietario los Beacons que tiene registrados para la tienda.

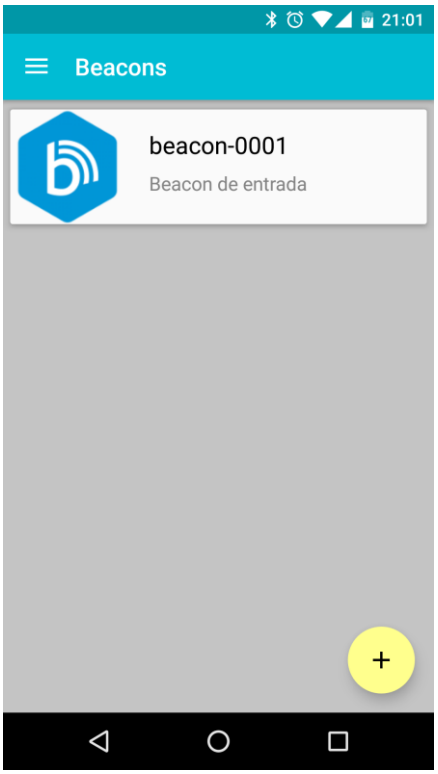


Ilustración 116 - Lista de Beacons

La pantalla **Usuarios** muestra al propietario la lista de usuarios registrados en su tienda junto con los puntos que tienen estos usuarios.

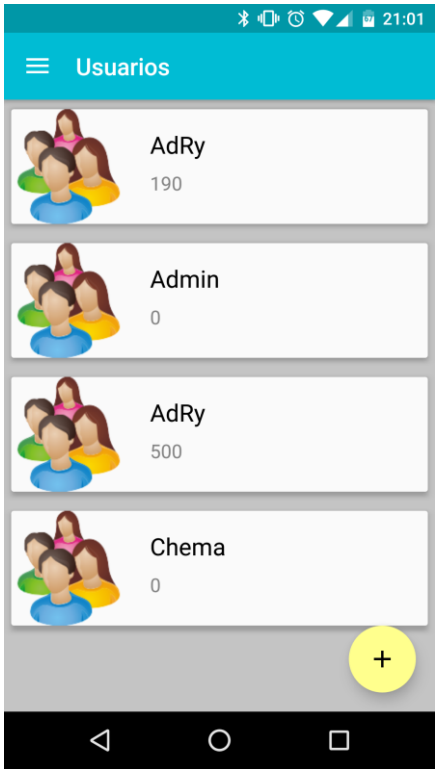


Ilustración 117 - Lista de usuarios

7. CONCLUSIONES

- ✓ De los objetivos establecidos al inicio del proyecto este sería el resultado después de terminar el proyecto:Conseguir que la solución sea lo más económica posible → La solución final del proyecto es bastante económica, ya que una tienda solo necesitaría comprar los **Beacons** y por nuestra parte solamente tendríamos que pagar la licencia de **Firebase**.

- ✓ Que los datos sean accesibles desde cualquier dispositivo (Almacenamiento en la nube) → Esto esta conseguido gracias a utilizar **Firebase** para el almacenamiento de la información.

- ✓ Que el usuario pueda utilizar la aplicación sin conexión puntualmente (Modo offline) → Es una de las características que nos ofrece **Firebase**.

- ✓ Que el establecimiento pueda establecer anuncios y asociarlos a un Beacon, y que lleguen al usuario al estar cerca de ese Beacon → Esto ha sido implementado y está funcionando correctamente.

- ✓ Que el cliente pueda registrar una compra de forma sencilla → El cliente podrá registrar sus compras en un establecimiento mediante **NFC o Código QR**.

- ✓ Que el cliente pueda consultar las promociones disponibles → Es una característica que se ha implementado en la aplicación.

Creo que los objetivos establecidos al inicio del proyecto se han cumplido, aunque durante la realización del proyecto fueron surgiendo nuevas ideas que se mostraran en el siguiente punto para futuras ampliaciones del proyecto.

En general estoy satisfecho con el resultado del proyecto ya que he podido aprender distintas tecnologías que conocía pero no había utilizado y he aprendido herramientas que directamente no conocía.

En cuanto a las críticas sobre el resultado final, me gustaría que hubiese tenido un resultado más cerca de un producto final, y con esto me refiero a que el diseño se acercase más a la de un producto terminado. Esto al final no ha sido posible por una cuestión de tiempo, ya que realizar el diseño de todas las pantallas de la aplicación me llevaría mucho tiempo y tampoco es que se me dé muy bien el diseñar aplicaciones.

8. SIGUIENTES PASOS

En este apartado se van a proponer posibles ampliaciones que se podrían llevar a cabo en un futuro en este proyecto. Los siguientes puntos surgen de ideas que se han ido ocurriendo durante el desarrollo del proyecto pero que no han podido llevarse a cabo por una cuestión de tiempo. También se proponen posibles mejoras de la implementación actual del proyecto.

8.1. NOTIFICACIONES SEGÚN CONSUMO

Actualmente se notifica solamente teniendo en cuenta la posición del usuario, que gracias a los Beacons, podemos saber cerca de qué tienda está y mostrarle un anuncio de esta tienda.

Habría que mejorar este sistema para que en el anuncio que se le muestra al usuario se tuviese en cuenta las categorías de productos que más consume y mostrarle un anuncio de esas categorías que tenga en esa tienda.

8.2. GUARDAR COMPRAS

Estaría bien que el usuario pudiese guardar los tickets de las compras que realiza en las tiendas directamente en la aplicación y que pudiese utilizarse posteriormente como justificante de la compra. Esto evitaría un problema que suele tener la gente a la hora de guardar los tickets de compra de los productos, ya que normalmente no saben dónde lo han dejado, perdiendo así la garantía del producto o la posibilidad de cambiarlo. Además, si este sistema fuese aceptado por los establecimientos, se podría proponer que los usuarios recibiesen en su aplicación directamente el ticket de compra ayudando a mejorar el medio ambiente gracias a la reducción de uso del papel.

8.3. ANDROID PAY

Android Pay es un proyecto que lanzó **Google** en septiembre de 2015. El objetivo de este proyecto es utilizar los dispositivos móviles para realizar los pagos en los establecimientos.

Debido a que **SmartFidelity** está centrado en unificar todas las operaciones de fidelización de los establecimientos, **Android Pay** sería una característica muy interesante a implementar en este proyecto, ya que con ello permitiríamos a todos los establecimientos que utilicen nuestro sistema el realizar pagos mediante este nuevo sistema, que seguro que en un futuro será ampliamente utilizado.

8.4. SEGURIDAD EN FIREBASE

Como ya se ha comentado en este documento, en el proyecto se está utilizando **Firebase** como sistema de almacenamiento de datos, autenticación… Pues bien, **Firebase** permite configurar la **seguridad de acceso a los datos** mediante un fichero de configuración en formato **JSON [62].** Para realizar la configuración tenemos que acceder al panel de administración que **Firebase** pone a nuestra disposición, y ahí encontraremos la opción para configurar la seguridad del acceso a los datos.

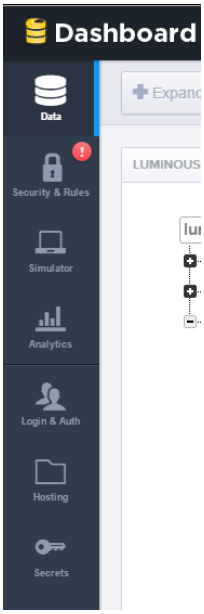


Ilustración 118 - Opciones del panel de administración de Firebase

En este proyecto se realizó una primera aproximación a lo que sería la configuración, pero habría que revisarla y adaptarla a las nuevas características que **Firebase** ha ido añadiendo para poder configurar esta seguridad. **Firebase** tiene muy bien documentado[35] cómo se debe realizar esta configuración y que posibilidades tenemos.

8.5. INTEGRACIÓN CON PHYSICAL WEB DE GOOGLE

Physical Web es un proyecto de **Google** del que ya se ha hablado en este documento. Este proyecto busca utilizar los **Beacons** para que transmitan una URL donde el establecimiento puede proporcionar al usuario información acerca de un producto o le puede permitir realizar una acción.

Debido a que los Beacons que se han utilizado en este proyecto son compatibles con **Physical Web,** en un futuro se podrían utilizar estos **Beacons** para adaptar este proyecto a esta nueva tecnología y ampliar las posibilidades que ofrecen los **Beacons.**

En este proyecto no se ha realizado porque esta tecnología todavía está en una fase inicial y sería recomendable esperar a que el proyecto estuviese un poco más maduro para que el desarrollo sea más sencillo y tenga mejor aceptación.

[35] Firebase - Understanding Security <https://www.firebase.com/docs/security/guide/understanding-security.html> [Disponible a 12 de marzo de 2016]

8.6. MEJORAR EL DISEÑO

Esto es un aspecto que ya he comentado antes, el diseño de la aplicación está bien adaptado a **Material Design**, pero el diseño de las vistas es el mismo en las listas y esto se podría mejorar y personalizar según los datos a mostrar.

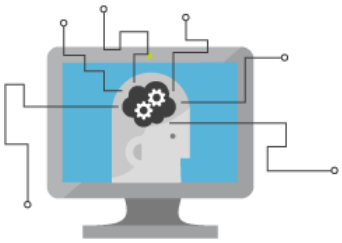
En realidad este trabajo no sería muy costoso, lo más costoso sería definir bien el diseño de cada pantalla y cada elemento y cambiar el diseño en las vistas que ya están creadas.

8.7. ALMACENAR IMÁGENES

Otro problema que no se ha atajado en este proyecto es el de almacenar las imágenes que se van a mostrar al usuario en las distintas pantallas de la aplicación (como por ejemplo en los **Anuncios**). Esta tarea se podría realizar o bien almacenando directamente las imágenes en **Firebase** (cosa que no sería lo más idóneo, aunque sí lo más sencillo) y, por otro lado, se podrían almacenar las imágenes en un servidor externo, y en **Firebase** simplemente almacenar la URLs de acceso a esas imágenes, luego con una libraría para Android como podrían ser **Picasso** o **Glide** se podría acceder a la imagen que se encuentra en el servidor de imágenes [63, 64].

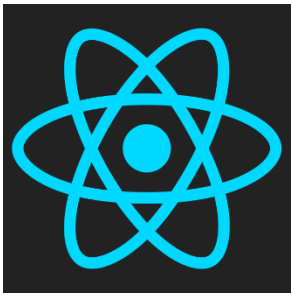
En definitiva, este es otro punto que habría que realizar en un futuro en este proyecto, ya que estaría muy bien que los propietarios de las tiendas pudiesen subir imágenes para sus anuncios o de sus tiendas por ejemplo.

8.8. MACHINE LEARNING



Otro punto en el que se podría trabajar es el de estudiar los datos de los usuarios para conocer mejor sus gustos y ayudar a las tiendas a acertar más en sus productos. Se podría crear un programa que estuviese continuamente analizando los datos almacenados en la BD para predecir posibles conductas futuras, nuevas tendencias y conocer mejor a los clientes. Además, se podrían hacer estudios concretos de cada usuario para mejorar las notificaciones que recibe y ofrecerle los productos en los que pueda estar más interesados en comprar, un estilo a lo que ya realizan grandes empresas como **Amazon o Google.**

8.9. MULTIDISPOSITIVO MEDIANTE REACT NATIVE



React Native es un Framework basado en **Javascript y React** propiedad de **Facebook [65].** El objetivo de este Framework es **facilitar la creación de aplicaciones multidispositivo**.

En un futuro se podría estudiar la posibilidad de utilizar **React Native** para desarrollar la aplicación y así tener disponible la aplicación compatible tanto para **Android** como para **IOS** o incluso para **Windows 10** si en un futuro **React Native** también lo soportase**.**

La ventaja que tendría realizar la aplicación con **React Native** es la **reutilización de casi el 90% del código** entre las distintas plataformas y además de poder realizar actualizaciones de la interfaz de la aplicación sin tener que subir una nueva versión a la Apple Store o Google Play, lo que supone un gran ahorro de tiempo, sobre todo en la plataforma de Apple.

Para conseguir ese casi 90% de reutilización de código, lo que hace este Framework es proporcionarnos componentes que son reutilizables para ambas plataformas (Android e IOS), por lo tanto, si nosotros hacemos uso de estos componentes que nos ofrece el Framework , solo escribiremos el código una vez y el Framework construirá la aplicación con los componentes nativos para cada versión. El 10% que no se puede reutilizar es por componentes que no están disponibles todavía o por algunas características que son exclusivas de cada plataforma, y en estos casos habrá que escribir el código para cada una de ellas. **React Native** tiene varios ejemplos en su documentación oficial, por ejemplo de cómo realizar una **ListView** con este Framework [66].

9. CONCLUSIÓN PERSONAL

Como conclusión personal de lo que ha significado para mí este proyecto, tengo que decir que sobre todo **ha sido un gran reto**, ya que aunque he tenido una asignatura en la carrera en la que hemos estudiado **Android**, en este proyecto se tocan muchos aspectos que no se han cursado en esa asignatura, como trabajar con **Material Design,** transferencia de información mediante **NFC**, trabajar con **Beacons**, **Bases de datos NoSQL en la Nube** (quizás el aspecto que más me ha costado), …

Cuando pensaba en qué proyecto me gustaría realizar al finalizar mi carrera, mi objetivo era que ese proyecto tenía que ser algo **nuevo, innovador**, que se pudiese llevar a cabo y que mejorara de una u otra forma la vida de las personas. Creo que estos objetivos los he cumplido con este proyecto, ya que considero que trabajando mucho más en él, puede llegar a ser un gran producto para la sociedad y tiene muchas posibilidades de ampliación (algunas ya mostradas en el punto anterior) y cada vez más gracias a la continua innovación en este campo.

No puedo terminar sin agradecer su tiempo y ayuda a mi mentor en este proyecto, el profesor José María Conejero, que ha sido de gran ayuda y ha estado siempre atento a la evolución del proyecto. También a todos aquellos profesores que me han enseñado tanto en la universidad y a los que les tengo mucho que agradecer.

10. BIBLIOGRAFÍA

1. Android (Wikipedia) <https://es.wikipedia.org/wiki/Android> [Disponible a 12 de diciembre de 2015]

2. Android consolidó su liderazgo en 2014 alcanzando el 81,5% de cuota de mercado <http://www.xatakandroid.com/sistema-operativo/android-consolido-su-liderazgo-en-2014-alcanzando-el-81-5-de-cuota-de-mercado> [Disponible a 12 de diciembre de 2015]

3. Android reina en España con casi un 90% de cuota de mercado <http://www.reasonwhy.es/actualidad/tecnologia/android-reina-en-espana-con-un-90-de-cuota-de-mercado-2015-09-03> [Disponible a 12 de diciembre de 2015]

4. Smartphone OS sales Market <http://www.kantarworldpanel.com/global/smartphone-os-market-share/> [Disponible a 12 de diciembre de 2015]

5. *Bluetooth (especificación)* <https://es.wikipedia.org/wiki/Bluetooth_(especificaci%C3%B3n)> [Disponible a 30 de noviembre de 2015]

6. *Bluetooth* <https://es.wikipedia.org/wiki/Bluetooth> [Disponible a 30 de noviembre de 2015]

7. Bluetooth de baja energía <https://es.wikipedia.org/wiki/Bluetooth_de_baja_energ%C3%ADa> [Disponible a 30 de noviembre de 2015]

8. *Bluetooth* <http://ipv6.com/articles/applications/Bluetooth.htm> [Disponible a 30 de noviembre de 2015]

9. *GAP* <https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap> [Disponible a 30 de noviembre de 2015]

10. *GATT* <https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt> [Disponible a 30 de noviembre de 2015]

11. *Bluetooth Low Energy SMP Pairing* <https://community.freescale.com/thread/332191> [Disponible a 30 de noviembre de 2015]

12. *Getting Started with Bluetooth Low Energy* <https://www.safaribooksonline.com/library/view/getting-started-with/9781491900550/ch01.html> [Disponible a 30 de noviembre de 2015]

13. *What is iBeacon? What are iBeacons?* <http://www.ibeacon.com/what-is-ibeacon-a-guide-to-beacons/> [Disponible a 13 de diciembre de 2015]

14. *Todo acerca de los "Beacons": usos y posibilidades que ofrecen* <*http://hipertextual.com/archivo/2014/10/todo-acerca-beacons/*> [Disponible a 13 de diciembre de 2015]

15. *McDonald's increases sales by beaming offers to smartphones in iBeacon pilot* <http://9to5mac.com/2014/12/18/mcdonalds-ibeacons/> [Disponible a 13 de diciembre de 2015]

16. iBeacon, otro ingrediente para la revolución del sector del comercio y la domótica <http://blogthinkbig.com/ibeacon-posicionamiento/> [Disponible a 13 de diciembre de 2015]

17. ¿Qué es la tecnología Beacon? Destapamos su potencial <http://www.compartirwifi.com/blog/que-es-la-tenologia-beacon-destapamos-su-potencial/ > [Disponible a 13 de diciembre de 2015]

18. NFC: qué es y para qué sirve <http://www.xataka.com/moviles/nfc-que-es-y-para-que-sirve> [Disponible a 13 de diciembre de 2015]

19. NFC en el móvil: cómo, cuándo y para qué <http://www.xatakandroid.com/mundogalaxy/nfc-en-el-movil-como-cuando-y-para-que> [Disponible a 13 de diciembre de 2015]

20. NFC: qué es y para qué sirve esta conexión inalámbrica <http://computerhoy.com/noticias/life/nfc-que-es-que-sirve-esta-conexion-inalambrica-24207> [Disponible a 13 de diciembre de 2015]

21. NFC en móviles <http://es.slideshare.net/eventoscreativos/nfc-en-mviles> [Disponible a 13 de diciembre de 2015]

22. NFC Forum Technical Specifications <http://members.nfc-forum.org/specs/spec_list/> [Disponible a 13 de diciembre de 2015]

23. NFC Forum Specification Architecture <http://nfc-forum.org/our-work/specifications-and-application-documents/specifications/> [Disponible a 13 de diciembre de 2015]

24. Estructura y seguridad de los QR Codes <http://elbauldelprogramador.com/estructura-y-seguridad-de-los-qr-codes/> [Disponible a 13 de diciembre de 2015]

25. Código QR <https://es.wikipedia.org/wiki/C%C3%B3digo_QR> [Disponible a 13 de diciembre de 2015]

26. Teorema CAP <https://es.wikipedia.org/wiki/Teorema_CAP> [Disponible a 20 de diciembre de 2015]

27. Bases de datos NoSQL: Llegaron para quedarse <http://basesdedatosnosql.blogspot.com.es/> [Disponible a 20 de diciembre de 2015]

28. El concepto NoSQL, o cómo almacenar tus datos en una base de datos no relacional <http://www.genbetadev.com/bases-de-datos/el-concepto-nosql-o-como-almacenar-tus-datos-en-una-base-de-datos-no-relacional> [Disponible a 20 de diciembre de 2015]

29. Bases de datos NoSQL. Elige la opción que mejor se adapte a tus necesidades <http://www.genbetadev.com/bases-de-datos/bases-de-datos-nosql-elige-la-opcion-que-mejor-se-adapte-a-tus-necesidades> [Disponible a 20 de diciembre de 2015]

30. NoSQL <https://es.wikipedia.org/wiki/NoSQL> [Disponible a 20 de diciembre de 2015]

31. Abandoning ACID in Favor of BASE <http://databases.about.com/od/otherdatabases/a/Abandoning-Acid-In-Favor-Of-Base.htm> [Disponible a 20 de diciembre de 2015]

32. ACID <https://es.wikipedia.org/wiki/ACID> [Disponible a 20 de diciembre de 2015]

33. Base de datos documental <https://es.wikipedia.org/wiki/Base_de_datos_documental> [Disponible a 20 de diciembre de 2015]

34. Material design <http://www.google.com/design/spec/material-design/introduction.html> [Disponible a 20 de diciembre de 2015]

35. Teoría del Color <https://es.wikipedia.org/wiki/Teor%C3%ADa_del_color> [Disponible a 20 de diciembre de 2015]

36. Material Palette <https://www.materialpalette.com/ [Disponible a 20 de diciembre de 2015]

37. Android Beacon Library <http://altbeacon.github.io/android-beacon-library/> [Disponible a 26 de diciembre de 2015]

38. Estimote Android SDK <https://github.com/Estimote/Android-SDK> [Disponible a 26 de diciembre de 2015]

39. Accent Systems BLE Beacons - Accent Systems <http://accent-systems.com/ble-beacons/> [Disponible a 26 de diciembre de 2015]

40. Android Studio + Cloud Endpoints + Objectify Persistence <http://rominirani.com/2015/03/11/android-studio-cloud-endpoints-objectify-persistence/> [Disponible a 26 de diciembre de 2015]

41. Gradle Tutorial : Part 7 : Android Studio + App Engine + Gradle <http://rominirani.com/2014/08/20/gradle-tutorial-part-7-android-studio-app-engine-gradle/> [Disponible a 26 de diciembre de 2015]

42. Gradle Tutorial : Part 8 : Gradle + App Engine + Endpoints + Android Studio <http://rominirani.com/2014/08/25/gradle-tutorial-part-8-gradle-app-engine-endpoints-android-studio/> [Disponible a 26 de diciembre de 2015]

43. Gradle Tutorial : Part 9 : Cloud Endpoints + Persistence + Android Studio <http://rominirani.com/2014/08/26/gradle-tutorial-part-9-cloud-endpoints-persistence-android-studio/> [Disponible a 26 de diciembre de 2015]

44. Retrofit - Getting Started and Create an Android Client <https://futurestud.io/blog/retrofit-getting-started-and-android-client> [Disponible a 26 de diciembre de 2015]

45. Objectify (Github Wiki) <https://github.com/objectify/objectify/wiki> [Disponible a 26 de diciembre de 2015]

46. Objectify Entities Relationships <https://github.com/objectify/objectify/wiki/Entities#relationships> [Disponible a 26 de diciembre de 2015]

47. Denormalizing Your Data is Normal <https://www.firebase.com/blog/2013-04-12-denormalizing-is-normal.html> [Disponible a 26 de diciembre de 2015]

48. Firebase - Pricing & Plans <https://www.firebase.com/pricing.html> [Disponible a 26 de diciembre de 2015]

49. Cloud datastore pricing <https://cloud.google.com/datastore/> [Disponible a 26 de diciembre de 2015]

50. Hold View Objects in a View Holder <https://developer.android.com/intl/es/training/improving-layouts/smooth-scrolling.html#ViewHolder> [Disponible a 24 de enero de 2016]

51. Strategy Design Pattern <http://www.javacodegeeks.com/2015/09/strategy-design-pattern.html> [Disponible a 31 de enero de 2016]

52. Strategy (patrón de diseño) <https://es.wikipedia.org/wiki/Strategy_(patr%C3%B3n_de_dise%C3%B1o)> [Disponible a 31 de enero de 2016]

53. Builder Design Pattern <http://www.javacodegeeks.com/2015/09/builder-design-pattern.html> [Disponible a 31 de enero de 2016]

54. Builder (patrón de diseño) <https://es.wikipedia.org/wiki/Builder_(patr%C3%B3n_de_dise%C3%B1o)> [Disponible a 31 de enero de 2016]

55. Android Beacon Library (Samples) <https://altbeacon.github.io/android-beacon-library/samples.html> [Disponible a 31 de enero de 2016]

56. Especificación de AltBeacon <https://github.com/AltBeacon/spec> [Disponible a 31 de enero de 2016]

57. Setting Up Bluetooth <https://developer.android.com/guide/topics/connectivity/bluetooth.html#SettingUp> [Disponible a 21 de febrero de 2016]

58. MaterialDrawer <https://mikepenz.github.io/MaterialDrawer/> [Disponible a 28 de febrero de 2016]

59. MaterialSpinner <https://github.com/ganfra/MaterialSpinner> [Disponible a 28 de febrero de 2016]

60. QrDroid <http://qrdroid.com/> [Disponible a 28 de febrero de 2016]

61. QrDroid for Android Developers <http://qrdroid.com/android-developers/> [Disponible a 28 de febrero de 2016]

62. Firebase - Understanding Security <https://www.firebase.com/docs/security/guide/understanding-security.html> [Disponible a 12 de marzo de 2016]

63. Picasso <https://square.github.io/picasso/> [Disponible a 12 de marzo de 2016]

64. Glide <https://github.com/bumptech/glide> [Disponible a 12 de marzo de 2016]

65. React Native <https://facebook.github.io/react-native/> [Disponible a 12 de marzo de 2016]

66. React Native – Ejemplo de ListView <https://facebook.github.io/react-native/docs/tutorial.html#listview> [Disponible a 12 de marzo de 2016]

67. Apps para almacenar tarjetas de fidelización <http://www.ticbeat.com/tecnologias/apps-para-almacenar-tarjetas-de-fidelizacion/> [Disponible a 10 de abril de 2016]

68. Nubbler diseña una app de fidelización al cliente que quiere llevar a 300 tiendas <http://www.laopiniondemalaga.es/malaga/2016/03/23/nubbler-disena-app-fidelizacion-cliente/837731.html> [Disponible a 10 de abril de 2016]

69. Alexander Osterwalder; Yves Pigneur (2011). *Generación de modelos de negocio.* http://goo.gl/CriNPZ