ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

DESARROLLO DE UNA APLICACIÓN DE COMERCIO ELECTRÓNICO BASÁNDONOS EN LA REUTILIZACIÓN DE UNA LIBRERÍA ANDROID PARA TRABAJAR CON BEACONS

DEVELOPMENT OF A SMART E-COMMERCE APPLICATION BASED ON THE REUSE OF AN ANDROID LIBRARY TO WORK WITH BEACONS

Realizado por
Juan Manuel Morales Joya

Tutorizado por
Mónica Pinto Alarcón

Departamento
Lenguajes y Ciencias de la Comunicación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DEL 2021

Fecha defensa: octubre de 2021

Resumen

Actualmente, el comercio electrónico, personalizado e inteligente, es uno de los sectores comerciales que más ha crecido como consecuencia de la evolución de distintas estrategias de marketing, la utilización de una tecnología aplicada a ventas y la confianza del usuario por el entorno web. Además, hoy en día, el IoT (Internet de las Cosas, *Internet Of Things*) está experimentando una gran evolución debido a las distintas facilidades que proporciona en diferentes aspectos de la vida cotidiana, como podrían ser las compras y ventas. Por todo esto, uno de los tipos de aplicaciones IoT que más auge está teniendo en los últimos años son las aplicaciones de comercio electrónico inteligentes.

La mayoría de las aplicaciones de comercio electrónico inteligentes basan su funcionamiento en el uso de Beacons, unos dispositivos IoT de muy bajo coste basados en la tecnología de BLE (Bluetooth de baja energía, *Bluetooth Low Energy*) que permiten identificar a los clientes situados en las proximidades de un determinado comercio. Utilizando estos dispositivos, un comercio puede mejorar considerablemente su programa de fidelización, ofreciendo beneficios a los clientes cuando más podrían estar interesados en recibirlos, es decir, cuando este se encuentra en los alrededores del negocio.

Este proyecto, enfocará el uso de estas aplicaciones de comercio electrónico al ámbito de las áreas comerciales donde se puedan instalar Beacons, de forma que cualquier cliente situado en las cercanías del área pueda recibir, a través de la aplicación, diferentes beneficios fruto del comercio inteligente personalizado. Para ello, se utilizará y evolucionará una librería reutilizable desarrollada en un TFG anterior que establece una comunicación básica entre los dispositivos IoT y la aplicación Android cliente que en este proyecto se va a desarrollar. Además de la aplicación mencionada, en este TFG se implementará una arquitectura completa con el desarrollo de un servidor de comercio inteligente donde residirá la lógica de la aplicación y unificará los servicios de comercio inteligente de cada negocio particular. Estos negocios a su vez deberán ofrecer la lógica de negocio que se aplicará para establecer la comunicación entre el servidor anteriormente mencionado y los servidores de cada negocio particular realizada a través de una interfaz común que también se diseñará en este proyecto.

Palabras claves: IoT, Beacons, Bluetooth, Comercio Electrónico, Android.

Abstract

Currently, customized and intelligent electronic commerce is one of the commercial sectors that has grown the most as a result of the evolution of different marketing strategies, the use of a technology applied to sales and user confidence in the web environment. In addition, today, the IoT (Internet of Things) is undergoing a great evolution due to the facilities it provides in different aspects of daily life, such as purchases and sales. For all this, one of the types of IoT applications that is having the most boom in recent years are smart e-commerce applications.

Most smart e-commerce applications base their operation on the use of Beacons, very low-cost IoT devices based on BLE (Bluetooth Low Energy) technology that allow the identification of customers located in the vicinity of a certain trade. Using these devices, a business can considerably improve its loyalty program by offering benefits to customers when they could be most interested in receiving them, that is, when they are around the business.

This project will focus the use of these electronic commerce applications in the field of commercial areas where Beacons can be installed, so that any client located around the area can receive, through the application, different benefits resulting from smart commerce personalized. For this, a reusable library developed in a previous TFG will be used and evolved that establishes a basic communication between the IoT devices and the client Android application that is going to be developed in this project. In addition to the aforementioned application, in this TFG a complete infrastructure will be implemented with the development of an intelligent commerce server where the application logic will reside and will unify the intelligent commerce services of each particular business. These businesses in turn must offer the business logic that will be applied to establish communication between the aforementioned server and the servers of each particular business that is carried out through a common interface that will also be designed in this project.

Keywords: IoT, Beacons, Bluetooth, e-Commerce, Android.

Índice

1

Introducción

1.1 Justificación

El presente proyecto se lleva a cabo con el fin de proporcionar una solución que ayude a entender que la inmersión del comercio electrónico como modelo de negocio es indispensable en la social actual. Cada vez son más frecuentes los negocios que incorporan transacciones comerciales y financieras realizadas mediante el procesamiento y la trasmisión de información a través de la red. Como consecuencia, cada vez son más las compañías que deciden incorporar diferentes tecnologías a sus comercios electrónicos, como pueden ser promociones, ofertas exclusivas o cualquier tipo de beneficio personalizado que pueda detectar clientes potenciales y mantenerlos como consumidores activos.

Por otro lado, el Internet de las Cosas (*Internet of Things* en inglés) está cambiando la forma de vida actual. La interconexión digital de objetos con internet hace que labores cotidianas resulten más sencillas ya sea optimizando distintos procesos de gestión, automatizándolos o monitorizándolos por control remoto.

Por ello, al ser temáticas innovadoras y actuales se ha decidido realizar este proyecto que reúne ambas tecnologías para desarrollar una solución que pueda ser aplicable en entornos reales de negocio, concretamente en áreas comerciales.

1.2 Motivación

El motivo principal por el cual se realiza este trabajo es la novedad, ya que, desde el punto de vista de los conocimientos adquiridos en el grado de Ingeniería Informática, los conceptos y tecnologías que se necesitan para la realización de este proyecto son totalmente desconocidos para el alumnado. Por tanto, la realización de este trabajo precisa de un tiempo prolongado de investigación, adaptación y asimilación de los conceptos y tecnologías empleados. Estos a su vez nutren de conocimientos necesarios para un próximo futuro laboral.

Otro motivo es la posibilidad de aportar al sector del comercio un producto útil que aporte beneficios tanto al usuario cliente como a las compañías mediante distintas herramientas de fidelización de clientes que hacen del comercio electrónico inteligente, un modelo de negocio en auge.

El último motivo por el que se realiza este trabajo es la posibilidad de utilizar y evolucionar una librería Android desarrollada en un TFG anterior para la comunicación entre los dispositivos IoT y la aplicación cliente que, además de otras partes de la arquitectura completa, se va a desarrollar en este proyecto. A partir del trabajo que se realizará, se contribuirá a la reutilización de código para trabajos futuros que pueden seguir la misma motivación de este proyecto o incluso enfocar la idea en otras temáticas diferentes.

1.3 Objetivos

El objetivo principal de este Trabajo de Fin de Grado es el desarrollo de una solución integral para el desarrollo de aplicaciones de comercio electrónico inteligente basada en el uso de dispositivos Beacon que incluya un conjunto significativo de las funcionalidades más comunes en este tipo de negocios. Esta solución ofrece tanto el desarrollo de una aplicación móvil como el diseño, desarrollo y despliegue de una arquitectura específica para este tipo de aplicaciones identificando las distintas capas que la forman. Estas secciones son la capa del cliente donde se encontraría la aplicación móvil, la capa del servidor de comercio inteligente y la capa con los servidores de acceso a cada negocio integrado en la solución. El desarrollo se realizará sobre una infraestructura común reutilizable. Para ello, se partirá de una librería reutilizable de funcionalidades muy limitadas implementada en otro TFG. Dicha librería reutilizable solo corroboraba la posibilidad de utilizar la tecnología en cuestión para ámbitos del comercio inteligente con una aplicación de prueba. Aun así, servirá de base para ciertas operaciones que ayudarán a lograr la meta principal del presente proyecto.

Por tanto, como se ha mencionado anteriormente, la finalidad esencial de este trabajo es el diseño e implementación de una solución completa de comercio inteligente. Esta solución estará dirigida a áreas comerciales de grandes dimensiones con negocios de distintos sectores (electrónica, moda, hogar, etc.) con el objetivo de unificar en un solo sistema servicios comunes de recomendación y personalización hacia el usuario cliente sobre los distintos comercios que dicha área pudiera contener. Esta incluirá las siguientes funcionalidades, a las que el usuario accederá a través de una aplicación móvil Android:

- Registro de usuarios clientes del área comercial con posibilidad de establecer diversas opciones de preferencias.
- Al acceder al área comercial y ser identificado por un Beacon, la aplicación envía al servidor del comercio inteligente la detección del usuario junto a sus datos de registro, preferencias seleccionadas y los Beacons disponibles para el usuario.
- La aplicación recibirá información por parte de los distintos negocios siguiendo las preferencias elegidas en los puntos anteriores. Esta información puede ser de diversos tipos, como podrían ser ofertas personalizadas, establecimientos disponibles según las preferencias, recomendaciones de productos o servicios, etc.

Para llevar a cabo el objetivo principal, este proyecto se apoya en la definición e implementación de una arquitectura software basada en capas. Una de estas capas es un servidor de comercio inteligente. En este servidor de aplicación reside la lógica de comercio inteligente detallada en la meta inicial. Como aporte, este servidor ofrecerá, a partir de una base de datos, la posibilidad de realizar el registro de cada negocio del área comercial interesado en integrarse al comercio inteligente incluyendo, si procediese, los Beacons

propios de ese negocio, el ámbito del negocio y el punto de acceso a la información del comercio.

La implementación de este servidor implica definir la interfaz entre la aplicación móvil y el servidor, pero también entre el servidor de comercio inteligente y cada uno de los negocios. Para ello, será necesario definir una capa con la lógica que cada negocio necesita implementar para integrarse en este sistema. Se definirán las interfaces proporcionadas y requeridas por dicho componente, delegando luego su implementación a cada negocio. Como prueba de concepto, se implementará una versión de este componente para un negocio concreto, añadiendo tanto la lógica implementada para el negocio (acorde a las restricciones de la aplicación) y el servidor necesario para ese negocio.

Como último fin, se revisará y evolucionará la infraestructura común existente para el desarrollo de aplicaciones de comercio electrónico inteligentes en Android. La aplicación que se desarrollará permitirá evaluar el grado de reutilización de la versión anterior de la infraestructura común que será actualizada o modificada de forma acorde a las nuevas necesidades identificadas.

1.4 Metodología

Este proyecto carece de equipo de trabajo, por tanto, utilizar una metodología ágil no sería la mejor forma de potenciar las ventajas que aporta la metodología al propio proyecto. Por ello, se ha optado por utilizar una metodología en espiral tradicional, donde las etapas constan del propio ciclo de vida del software: análisis, diseño, implementación y pruebas.

Las fases de proyecto que se han identificado son:

- **Análisis**. La primera parte es una etapa de análisis. En esta fase se estudian los objetivos a conseguir, el alcance del proyecto y la viabilidad de este. Por consiguiente, en esta etapa se obtienen los requisitos o necesidades que la aplicación debe satisfacer. En este primer punto, es importante dividir la parte reutilizable de la aplicación, ya sea de la infraestructura común reutilizable inicial como lo que se añada posteriormente, de la parte específica del entorno.
- **Diseño**. Como seguimos una metodología en espiral, los requisitos pueden ampliarse o modificarse entre distintas fases. Por ello, esta etapa comienza con la depuración de los requisitos del sistema. Posteriormente, se comienza con la fase de modelado y diseño de distintos diagramas con el objetivo de representar los aspectos del sistema utilizando diversos mecanismos de abstracción. A partir de estos modelos se aporta la especificación y comprensión necesaria para poder abordar el desarrollo de la propia aplicación. Para finalizar esta etapa se diseñarán las distintas interfaces de usuario que satisfagan los requisitos establecidos.
- **Implementación.** En este punto se verificarán los requisitos y diagramas diseñados con el objetivo de asegurar la calidad formal de los distintos modelos y poder comenzar a desarrollar el producto. Luego de esta verificación se procederá al desarrollo de la solución de comercio electrónico propuesta.
- **Pruebas.** Finalmente, se procederá a la realización de distintas pruebas con el propósito de realizar un software de calidad y poder detectar cualquier error que se haya podido tener en etapas anteriores. En caso de detección de errores, estos serán analizados mediante un estudio de los procesos realizados en fases anteriores, con el

fin de detectar los posibles fallos y poder solventarlos.

1.5 Estructura de la memoria

Esta memoria está estructurada en 5 capítulos y 2 apéndices que abarcan todo el contenido fundamental para la comprensión del proyecto implementado. A modo de introducción, se puede resumir el contenido de cada capítulo de la siguiente manera:

1. **Introducción.** Capítulo presente donde se expone la justificación, la motivación, la metodología que se seguirá y los objetivos del proyecto que se detalla.
2. **Tecnologías.** En este capítulo se presentan las tecnologías utilizadas, resaltando los rasgos característicos y el motivo de la elección de estas.
3. **Análisis de requisitos y Diseño.** En este apartado, se detallarán todas las funciones que realizará el sistema. Además, se analizará las opciones disponibles de implementación para el proyecto que hay que construir, así como decidir la estructura general del mismo.
4. **Implementación y pruebas.** Tras la fase de diseño del sistema, este capítulo detalla cómo se ha implementado el proyecto y qué tipo de ensayos se han realizado para corroborar que funciona de manera correcta.
5. **Conclusiones y trabajo futuro.** En este capítulo se exponen los resultados finales obtenidos y qué situaciones se han presentado durante el proceso. También se documentará las posibles continuaciones que podría tener este Trabajo de Fin de Grado.

- Apéndice A. **Manual de instalación.** Este apéndice contiene la información necesaria para poner en funcionamiento todo el sistema implementado.

- Apéndice B. **Manual de usuario.** Mediante este apartado, los usuarios clientes podrán visualizar cómo puede utilizarse cada funcionalidad aportada por la aplicación Android desarrollada.

2

Tecnologías

En el siguiente capítulo se detallarán las tecnologías utilizadas para llevar a cabo el proyecto que se expone en la presente memoria. Se especificarán las características principales de cada tecnología, el motivo de su elección y qué aporta para satisfacer los objetivos propuestos en este Trabajo de Fin de Grado.

2.1 Android

Figura 2.1 Logo Android de: https://blog.google/products/android/

Es un sistema operativo empleado en dispositivos móviles que fue creado, en sus orígenes, por Android Inc., compañía que fue absorbida en 2005 por Google. Esta empresa es la actual responsable del continuo desarrollo que el sistema operativo mantiene de manera libre y de código abierto.

Android basa su núcleo en Linux, un software libre que se centra, a su vez, en Unix mediante máquinas virtuales Dalvik (en sus inicios) o ART (en versiones actuales).

Para el desarrollo de la aplicación cliente en Android, se ha utilizado el entorno Android Studio. Este es el entorno de desarrollo integrado (IDE) oficial para la implementación de aplicaciones en Android, se centra en IntelliJ IDEA y además de ser uno de los editores de código más potentes, dispone de características que mejoran la productividad de desarrollo. Algunas de estas funciones son [1]:

- Herramientas Lint para determinar problemas de rendimiento o compatibilidad de versiones.
- Compatibilidad con C++ y NKD.
- Integración con plantillas de código o GitHub para compilar funciones comunes en distintas aplicaciones.

- Compatibilidad con Google Cloud Platform que, a su vez, ayuda a la integración con otras aplicaciones como App Engine.

2.2 Beacon

Figura 2.1 Dispositivo beacon de: https://www.ticbeat.com/tecnologias/que-es-beacon-y-en-que-consiste/

Es una tecnología que centraliza su funcionamiento en el principio de transmisión de datos Bluetooh Low Energy (BLE) facilitando una comunicación automatizada entre unos dispositivos IoT llamados balizas (Beacons, en inglés) y receptores como podrían ser los dispositivos móviles. El propósito principal de esta tecnología es la difusión de mensajes (broadcasting, en inglés) entre la baliza electrónica y los distintos receptores de la red que sean capaces de utilizar el protocolo Bluetooth 4.0 y alcancen las ondas emitidas en la comunicación [2].

Los datos que los dispositivos emiten a los distintos dispositivos se pueden enviar en distintos formatos. La posibilidad de utilizar los distintos formatos de datos depende del protocolo utilizado. Los más conocidos son iBeacon que es creado por la compañía Apple, Eddystone creado por Google y Altbeacon implementado por Radius Networks [3].

2.3 Node.js

Figura 2.3 Logo Node.js de: https://nodejs.org/es/about/resources/

Para el desarrollo de la arquitectura de este proyecto se empleará el lenguaje Node.js, uno de los lenguajes más utilizados para la implementación de servidores por sus útiles recursos y posibles integraciones con distintas tecnologías.

Node.js es un entorno de ejecución multiplataforma de código abierto para desarrollar aplicaciones del lado del servidor construida sobre el motor JavaScript de Google Chrome V8.

A partir de Node.js se pueden desarrollar aplicaciones de red escalables y rápidas, escritas en JavaScript, utilizando un modelo de entrada y salida (E/S) que recibe eventos sin bloquearse. Algunas de las características más importantes son [4]:

- Los servidores implementados en Node.js son asíncronos y controlados por eventos, por lo que nunca esperarán a que una API devuelva los datos, sino que, mediante un mecanismo de notificación de eventos, obtendrán respuesta a las llamadas asíncronas que realicen.
- El motor JavaScript V8 hace que la biblioteca Node.js sea de ejecución de código muy rápida.
- Las aplicaciones generan los datos en trozos y no los almacenan en búfer.
- Suministra una biblioteca que contiene varios módulos de JavaScript que facilitan potencialmente el desarrollo de aplicaciones web. Uno de los más importantes para el presente proyecto es el módulo Express, esencial para la creación de servidores HTTP.

2.4 Webhook

Es un modo de proporcionar información a tiempo real entre aplicaciones a través del protocolo HTTP. Webhook es utilizado con frecuencia para la notificación de eventos en el servidor.

Informalmente, los webhooks se denominan *"API inversas"* ya que proporcionan características similares a las de las API, pero del lado del servidor hacia el cliente. Una de sus características diferenciadores con respecto a las API es que la trasmisión se realiza de manera asíncrona, sin necesidad de esperar la respuesta del servidor.

En este proyecto, se utilizará Webhook para realizar las comunicaciones entre los servidores de los distintos negocios y el servidor central y poder trasmitir los datos necesarios que serán mostrados en la aplicación.

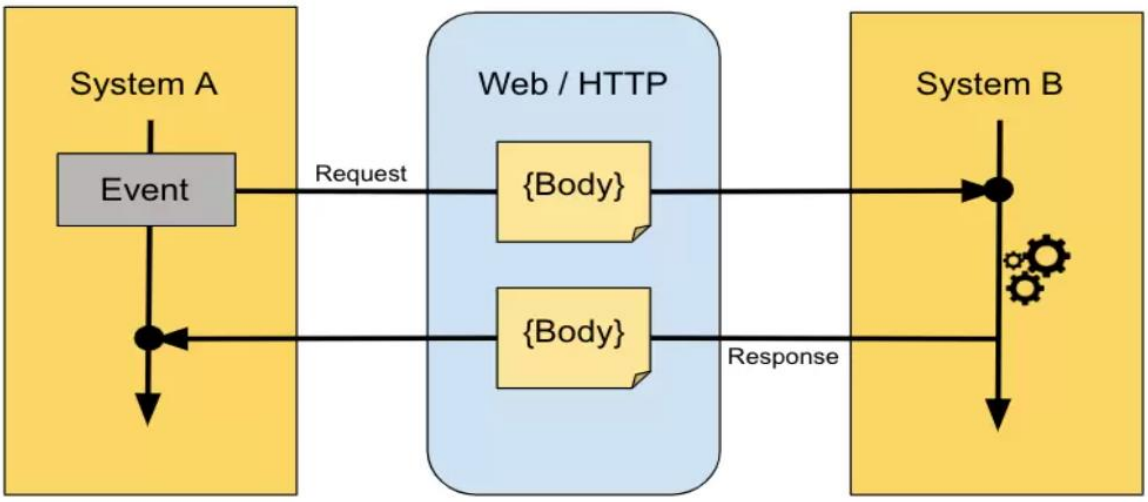


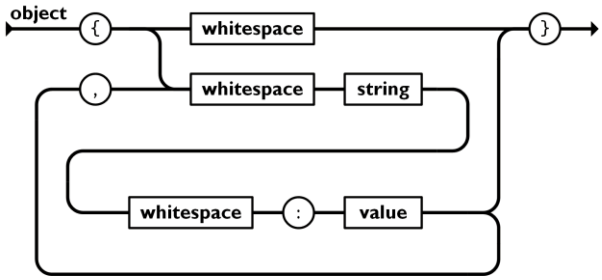
Figura 2.4 Flujo Webhook de envio-respuesta de: https://www.hebergementwebs.com/noticias/explicacion-de-los-webhooks-que-son-y-como-usarlos

2.5 JSON

Son las siglas de JavaScript Object Notation, es un formato ligero para almacenar y transportar datos de manera sencilla, independientemente del lenguaje de programación utilizado. Json está constituído por dos estructuras:

- Una colección de pares de nombre y valor.
- Una lista ordenada de valores.

Estas estructuras son universales y todos los lenguajes de programación las implementan de alguna forma. En este proyecto, se utilizará como formato para trasmitir datos entre los distintos servidores y entre el servidor central y la aplicación cliente.



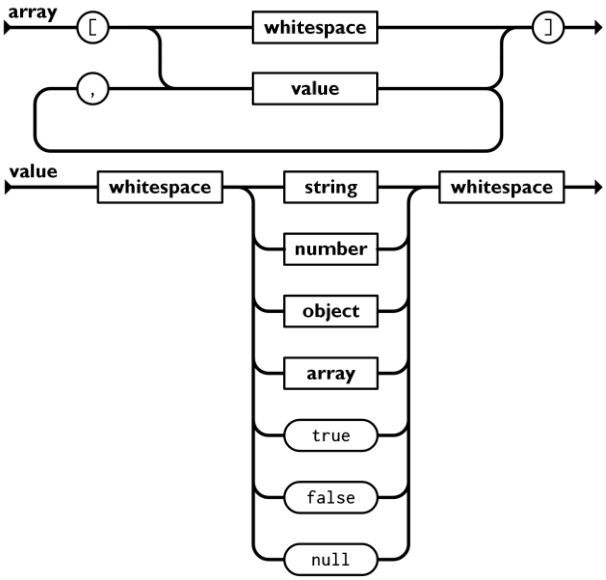


Figura 2.5 Formato Json de: https://www.json.org/json-es.html

2.6 Mongo DB

Figura 2.6 Logo mongo DB de: https://www.mongodb.com/brand-resources

Es una base de datos NoSQL orientada a documentos, es decir, mongo DB no guarda sus datos en registros ni tablas, sino en documentos que son almacenados en formato BSON, una representación binaria de JSON. Las características más llamativas de mongo DB son la velocidad, el balance perfecto entre rendimiento y funcionalidad, la capacidad de hacer consultas ad hoc, la indexación, la replicación y la ejecución de JavaScript del lado del servidor.

Al contrario de las bases de datos relacionales, con mongo DB no es necesario seguir un esquema ya que documentos de una misma colección pueden tener esquemas diferentes.

Por todo esto y por mantener la relación entre el formato de los datos que se van a trasmitir en este proyecto y los que se guardan en este tipo de bases de datos (formato JSON), se utilizará mongo DB para almacenar, en el servidor central de la aplicación, tanto los datos de los usuarios registrados como los negocios que quieran participar en el sistema.

2.7 Retrofit

Figura 2.7 Logo Retrofit de: https://www.programaenlinea.net/consume-una-api-procesa-la-respuesta-retrofit/

Es un cliente de servidores REST para Java y Android creado por Square que permite hacer diferentes peticiones al servidor (GET, POST, PUT, etc.) y tramitar distintos tipos de parámetros para obtener automáticamente la respuesta en un tipo específico de datos.

En el presente proyecto se utilizará Retrofit para realizar las distintas peticiones desde la aplicación Android al servidor de esta. Para poder utilizar este cliente en Android, se ha de configurar ciertos ficheros del proyecto como son *'build.gradle'* y *'android.manifest'* para poder acceder al servicio REST que se aloja en un servidor de Internet.

2.8 Firebase

Figura 2.8 Logo Firebase de: *https://firebase.google.com/brand-guidelines?hl=es*

Firebase es una plataforma en la nube de Google para la creación y gestión de aplicaciones para la web y móvil (IOS y Android). En sus inicios, Firebase comenzó siendo una base de datos en tiempo real, sin embargo, actualmente posee muchas más funcionalidades que facilitan el desarrollo de aplicaciones.

Para este proyecto, será necesario integrar Firebase para poder transmitir la información obtenida en el servidor de la aplicación, a través del webhook, hasta la propia app Android. Para ello, se han utilizado dos funcionalidades principales de Firebase. Estas son:

- **Realtime Database.** Es una de las herramientas más destacadas de la plataforma, ya que gracias a estas bases de datos a tiempo real se puede alojar datos de manera temporal, manteniéndolos actualizados en todo momento. Estas bases de datos residen en la nube, son No SQL y almacenan los datos en formato JSON,

manteniendo la relación de tipo con todo el proyecto. Concretamente, esta base de datos se utilizará para mantener de manera temporal (mientras el usuario esté en el interior del Área Comercial) los datos referentes al usuario en relación con los beneficios obtenidos, ya que será necesario recuperar esta información al acceder a distintas actividades de la aplicación Android.

- **Cloud Messaging.** Su función es el envío de notificaciones y mensajes a distintos usuarios en tiempo real y a través de varias plataformas. En cuanto a este proyecto se refiere, se utilizará esta función para el envío de los beneficios desde el servidor de la app a la aplicación Android cliente. Además del envío de esa información, se utilizará el mismo recurso para que el usuario que recibe el mensaje tome también una notificación en tiempo real (*push notification* en inglés) que avise de la recepción.

2.9 API REST

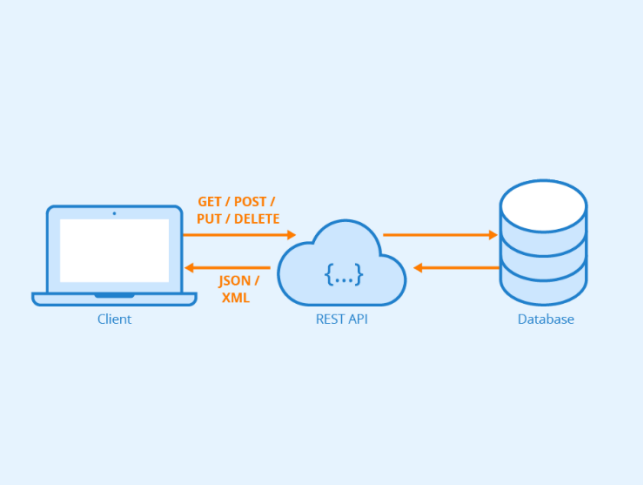


Figura 2.9 API REST de https://www.astera.com/es/tipo/blog/definici%C3%B3n-de-la-API-de-descanso/

Una API es una serie de protocolos y definiciones que sirven para implementar e integrar software de aplicaciones. Es una relación entre el sistema que provee la información y el usuario cliente, donde a partir de una llamada se obtiene una respuesta con los datos deseados.

La transferencia de estado representacional (REST de la abreviatura en inglés) es un conjunto de límites de arquitectura que abarca las siguientes características:

- Un protocolo cliente/servidor sin estado donde cada mensaje HTTP contiene toda la información necesaria para procesar la petición.
- Un conjunto de operaciones bien definidas que operan sobre toda la información disponible. Las más destacadas son GET, POST, PUT y DELETE.
- Sistema universal en torno a una sintaxis inequívoca que direcciona cada recurso de forma única a través de un identificador de recursos uniforme (URI de la abreviatura en inglés) [7].

2.10 Visual Studio Code

Figura 2.10 Logo Visual Studio Code de: https://code.visualstudio.com/

Es un editor de código fuente multiplataforma desarrollado por Microsoft que, entre otras cosas, incluye apoyo a la depuración de código, integración con Git, refactorización de código y autocompletado inteligente de código.

Aunque para la implementación de clientes y servidores de aplicaciones Node.js no se necesita ningún entorno concreto, utilizando Visual Studio Code se pueden emplear los distintos beneficios que aporta para realizar el trabajo de manera rápida, dinámica y sencilla. Como ejemplo, uno de los beneficios que ayudará al desarrollo de este proyecto será el terminal integrado en el propio entorno, que simplifica las tareas de ejecución del servidor.

2.11 Postman

Figura 2.11 Logo Postman de: https://www.postman.com/company/press-media/

Es una plataforma cliente API que se utiliza como ayuda al desarrollo de API. Postman ofrece funciones que facilitan la creación de una API enviando de forma sencilla solicitudes de distintos tipos como SOAP, REST y GraphQL.

Esta aplicación se utilizará para realizar distintas pruebas, en fase de desarrollo, sobre las distintas API REST que la arquitectura de este proyecto posee, para solicitar peticiones que facilitan la visualización de resultados sin necesidad de implementar un cliente concreto.

3

Análisis de requisitos y Diseño

Para desarrollar software de calidad es fundamental realizar un proceso de búsqueda para identificar qué se necesita. Por ello, en el presente capítulo se documentarán los requisitos obtenidos tras su respectivo análisis. Un requisito es una función (requisito funcional), característica, condición o capacidad (requisito no funcional) que un sistema debe realizar, satisfacer o gestionar.

Además, para estructurar la arquitectura del sistema junto a su entorno tecnológico y para especificar de una forma detallada los componentes del propio sistema, se integrarán en este capítulo distintos diagramas pertenecientes al diseño del proyecto.

Dado este punto, para facilitar la visualización de la arquitectura que se analizará, diseñará y posteriormente se implementará, se ha decidido añadir un diseño gráfico de la misma. Sobre este diseño se detallarán cada una de las partes del sistema.

3.1 Análisis de requisitos

Tal y como se especificó anteriormente en el apartado de objetivos (ver apartado 1.3) en este proyecto existen diversos objetivos a llevar a cabo. Por ello, es razonable que para capturar los requisitos necesarios que abarcan toda la arquitectura, será necesario dividirlos por las distintas metas existentes.

A su vez, se seccionarán los requisitos en dos tipos, requisitos funcionales, que son los relacionados con la operatividad técnica del sistema y requisitos no funcionales, que especifican criterios relativos a comportamientos específicos o condiciones necesarias que aseguran el buen funcionamiento del sistema.

3.1.1 Arquitectura del sistema

Un patrón de diseño arquitectónico habitual en el desarrollo de aplicaciones Web y aplicaciones móvil es el de arquitectura en N capas, donde habitualmente las capas a desarrollar son al menos tres: la capa del cliente, la capa del servidor y la capa de la base de datos. Por ese motivo, en este TFG empezamos por el diseño de la arquitectura en capas de nuestra solución, incluso antes de realizar el análisis de los requisitos, ya que estos serán generados a partir del diseño de la arquitectura. En la siguiente imagen, se podrá visualizar la arquitectura del proyecto, donde se aprecian las capas del sistema bien diferenciadas:

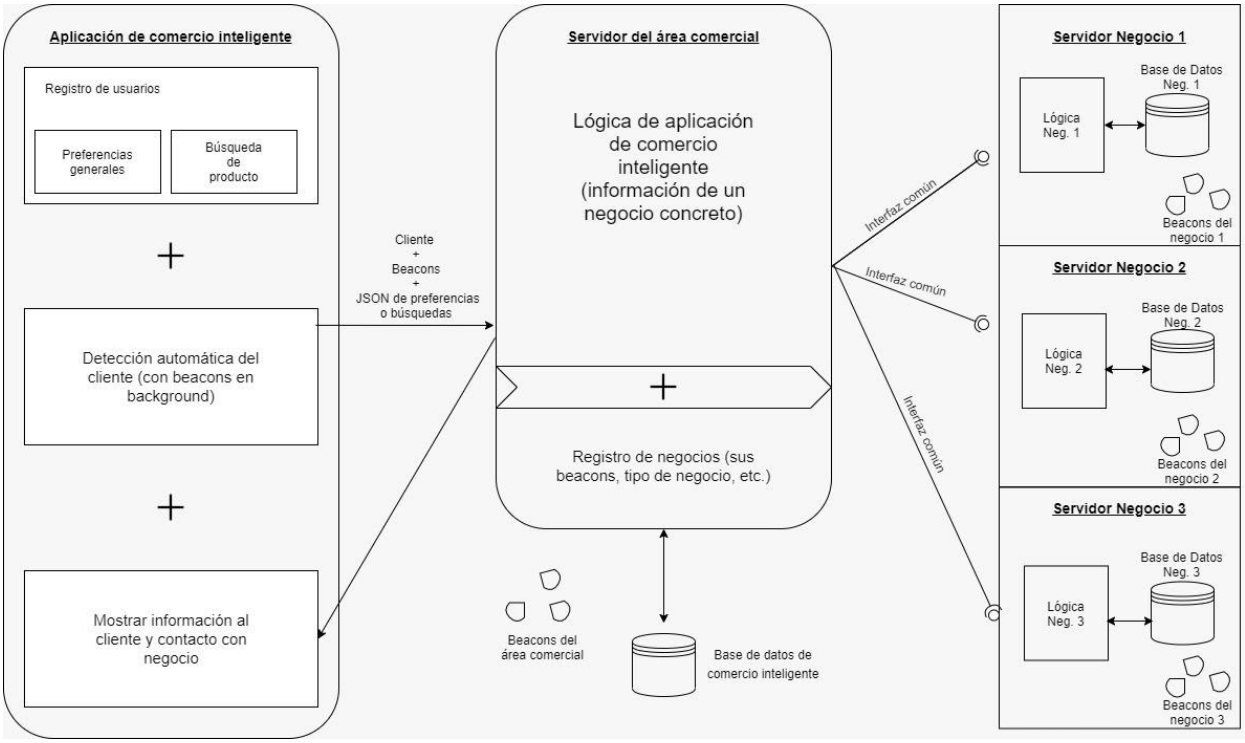


Figura 3.1 Diseño gráfico de la arquitectura del sistema

Seccionando la arquitectura del proyecto (ver figura 3.1), si se visualiza la parte derecha de la imagen, se puede ver la capa de la lógica de cada negocio que se integra al sistema. En ella, se definirán las reglas que cada comercio impondrá sobre sus datos. Estas reglas se gestionarán para adaptarlas a las restricciones impuestas por el sistema mediante la creación de servidores que monitorizarán las distintas funcionalidades que se implementarán posteriormente.

La sección anterior, intercambiará información con la capa situada en el centro de la imagen. Esta es la capa del servidor de comercio inteligente, donde se añadirán las funcionalidades principales de la arquitectura del sistema. Para poder establecer un intercambio de datos con los servidores de acceso a cada negocio de manera adecuada, como se puede apreciar en la imagen, será necesario establecer una interfaz común. Además, en esta sección central se monitorizarán los distintos dispositivos Beacons que puedan ser detectados y, se gestionarán y almacenarán diferentes datos, como son los referentes a los usuarios que deseen utilizar los servicios que proporciona la arquitectura o los negocios que quieran integrarse al sistema.

Los datos que se gestionan y almacenan en la capa de servidor de comercio inteligente serán enviados desde la sección situada a la izquierda de la figura 3.1. Esta es la capa del cliente donde se encontraría la aplicación móvil. A partir de ella, los usuarios podrán realizar diversas acciones disponibles en el sistema y aportar los datos necesarios para que dichos hechos se ejecuten de manera correcta. También, como se aprecia en la imagen, esta capa será la encargada de detectar automáticamente al cliente a través de los dispositivos Beacons alojados en el Área Comercial.

Por último, la capa cliente, mediante la aplicación Android, mostrará la información de comercio inteligente que será enviada desde la capa de servidor central que, a su vez, fue generada y enviada desde la capa de la lógica de negocio.

3.1.2 Requisitos de la aplicación Android

- **Requisitos Funcionales.**
  - **R.F.1. Registrar usuario.** El usuario debe poder registrarse, añadiendo su nombre, correo electrónico y preferencias comerciales.
  - **R.F.2. Validar acceso.** El usuario deberá autenticarse para acceder a la aplicación. La identificación de usuario se realizará a partir de su correo electrónico y contraseña asociada al registro (R.F.1).
  - **R.F.3. Editar preferencias.** El usuario puede editar sus preferencias comerciales.
  - **R.F.4. Mostrar ofertas personalizadas.** El usuario puede obtener ofertas de productos si su situación comercial se lo permite.
  - **R.F.5. Mostrar recomendaciones.** El usuario puede obtener recomendaciones de productos si su situación comercial se lo permite.
  - **R.F.6. Volver al beneficio.** El usuario puede volver a su beneficio obtenido (oferta o recomendación) siempre que permanezca en las instalaciones del Área Comercial.
  - **R.F.7. Registrar citas.** El usuario puede indicar que está esperando para mantener su cita previamente acordada con el negocio en cuestión.
  - **R.F.8. Mostrar información de citas.** El usuario puede obtener información sobre el estado de su cita.
  - **R.F.9. Mostrar obsequio.** El usuario puede recibir un café gratis si llega a su cita antes de que pueda ser atendido.
  - **R.F.10. Visualizar mapa.** El usuario puede visualizar un mapa del Área Comercial si accede a la aplicación sin estar en su interior.

- **Requisitos no funcionales.**
  - **R.N.F.1. Plataforma.** La aplicación debe ser desarrollada en Android.
  - **R.N.F.2. Interfaz.** La interfaz tiene que ser interactiva, amigable y fácil de usar.
  - **R.N.F.2. Hardware.** Para utilizar la aplicación se deberá utilizar un dispositivo móvil con conexión a internet y localización y sistema operativo Android con versión 5.0 o posterior (SDK mínimo 21).
  - **R.N.F.3. Seguridad.** El sistema debe cumplir con la Ley Orgánica de Protección de Datos.
  - **R.N.F.4. Calendario.** El sistema debe controlar el calendario actual del sistema para gestionar las citas.

3.1.3 Requisitos del servidor de la aplicación

- **Requisitos funcionales.**
  - **R.F.1. CRUD usuarios.** El servidor implementa la opción de crear, mostrar, editar y eliminar usuarios.
  - **R.F.2. CRUD negocios.** El servidor implementa la opción de poner crear, mostrar, editar y eliminar negocios que quieran participar en la arquitectura que se desarrolla.
  - **R.F.3. Iniciar Webhook.** El servidor tiene que iniciar de manera automática el proceso de obtención de beneficios.
  - **R.F.4. Obtener ofertas personalizadas.** El servidor puede obtener ofertas personalizadas de clientes desde el servidor del negocio que dispone del producto.
  - **R.F.5. Obtener recomendaciones.** El servidor puede obtener recomendaciones de productos desde el negocio que dispone de los productos ofrecidos.
  - **R.F.6. Obtener información de citas.** El usuario puede obtener información sobre el estado de su cita.
  - **R.F.7 Enviar Notificación.** El servidor debe enviar la información obtenida desde el servidor del negocio hacia el dispositivo móvil a través de los servicios de Firebase (Ver apartado 2.8).

- **Requisitos no funcionales.**

  - **R.N.F.1. Plataforma.** El servidor es implementado en Node.js.
  - **R.N.F.2. Hardware.** Se deberá de disponer un equipo con conexión a internet, Node.js y algún editor de código como es Visual Studio Code.
  - **R.N.F.3. Seguridad.** El servidor debe encriptar las contraseñas de los usuarios registrados.
  - **R.N.F.4. Calendario.** El sistema debe controlar el calendario actual del sistema para gestionar las citas.

3.1.4 Requisitos de la lógica y servidor del negocio

- **Requisitos funcionales.**
  - **R.F.1. Buscar cliente.** El sistema debe buscar al cliente que utiliza la aplicación en el registro del negocio en cuestión.
  - **R.F.2. Buscar productos relacionados.** El sistema debe buscar productos relacionados con los que el usuario ha comprado, las preferencias elegidas y los disponibles en el negocio.
  - **R.F.3. Devolver Webhook.** El sistema debe tras efectuar las acciones pertinentes, mandar una petición al servidor de la aplicación con la información requerida.
  - **R.F.4. Generar ofertas personalizadas.** El sistema debe generar una oferta personalizada según las preferencias y compras del cliente y los productos disponibles en el negocio.
  - **R.F.5. Generar recomendaciones.** El sistema debe generar recomendaciones según las preferencias y compras del cliente y los productos disponibles en el negocio.

- **R.F.6. Generar información de citas.** El sistema debe generar la información sobre la posible cita del usuario según si tiene cita registrada en el negocio, si ha llegado antes de la hora determinada, si ha llegado a tiempo o si ha llegado tarde.

- **Requisitos no funcionales.**

  - **R.N.F.1. Plataforma.** El servidor es implementado en Node.js.
  - **R.N.F.2. Hardware.** Se deberá de disponer un equipo con conexión a internet, Node.js y algún editor de código como es Visual Studio Code.
  - **R.N.F.3. Seguridad.** El servidor debe asegurar el envío de datos con una clave API.
  - **R.N.F.4. Calendario.** El sistema debe controlar el calendario actual del sistema para gestionar las citas.

3.2 Diseño

En esta fase del proyecto se estudiarán las posibles opciones de implementación para el proyecto a construir y se identificará la estructura general del mismo con el propósito de definir con suficiente detalle el sistema como para permitir su interpretación y futura realización física (implementación). Para ello se aplicarán ciertas técnicas y principios que estarán presentes en este capítulo.

3.2.1 Conexión Cliente-Servidor

La red cliente-servidor es una red de comunicaciones en la cual los clientes están conectados a un servidor, en el que se centralizan los diversos recursos y aplicaciones con que se cuenta; y que los pone a disposición de los clientes cada vez que estos son solicitados [6]. Por ello, en el servidor se gestionan todas las funcionalidades disponibles para el cliente.

En este proyecto en concreto, los servidores tendrán una arquitectura basada en API que siga un conjunto de límites REST, es decir, que disponga de una interfaz de programación de aplicaciones (API de la abreviatura en inglés) que permita la interacción con los servicios web de RESTful (ver apartado 2.9).

3.2.2 Patrón Modelo Vista Controlador

La finalidad de este modelo creado por Trygye Reenskaug en 1979 es poder organizar la información, la lógica del sistema y la interfaz que aporta al cliente en términos de modelo, controlador y vista respectivamente. En definitiva, este patrón sirve para clasificar y segmentar las partes del proyecto y así reducir la complejidad de este, aportando mantenimiento y flexibilidad en el código implementado.

En este proyecto se ha mantenido el patrón MVC jerárquicamente desde la codificación en la lógica y servidor del negocio, en el servidor central de la aplicación y, por supuesto, en la aplicación Android.

Un proyecto Android es sencillo de organizar bajo este patrón ya que el modelo correspondería a las clases de datos, el controlador las propias actividades Android manejadas y el modelo se identificaría con los diseños (layout en inglés).

Para poder visualizar de manera rápida y eficaz los distintos módulos, funciones y relaciones existentes en la aplicación Android se ha realizado un diagrama de clases del sistema.

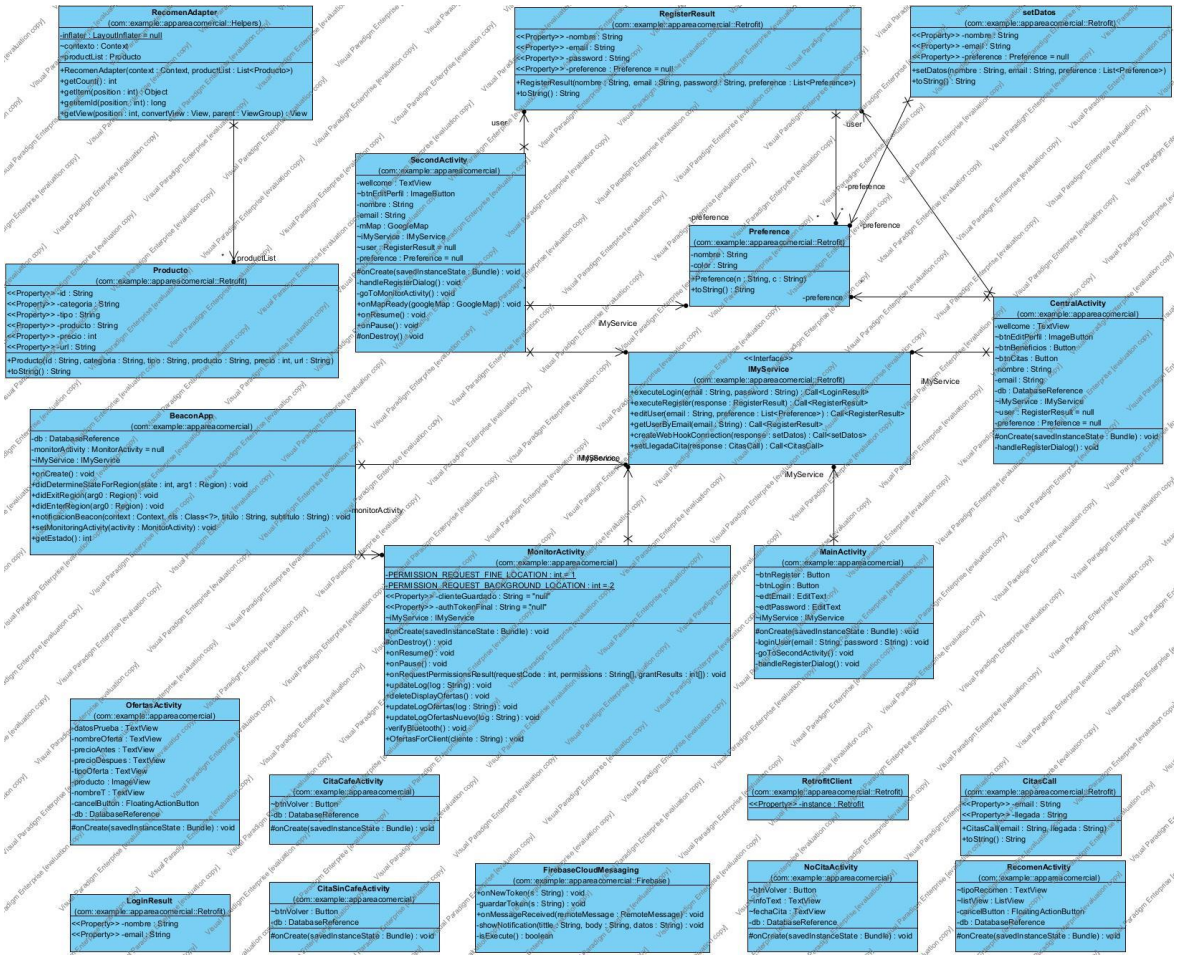


Figura 3.2 Diagrama de clases de la Aplicación Android generado en *Visual Paradigm Enterprise*

De esta forma, el proyecto, bajo la carpeta ***app*** dispondrá de todos los módulos implementados tanto en Java, para el modelo y controlador, como en XML para las vistas (ver figura 3.3), debito a esto se podrá acceder, delimitar y modificar si fuese necesario cualquier parte de la aplicación Android.

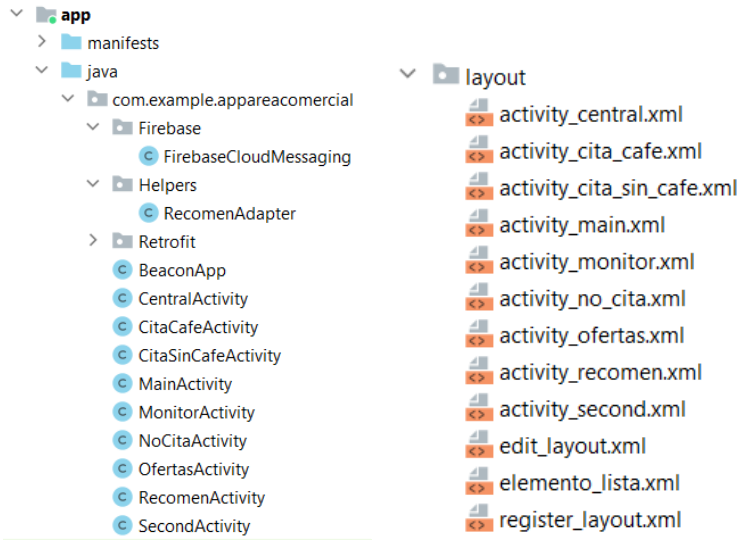


Figura 3.3 Jerarquía de archivos de la app Android

En los servidores implementados, tras las distintas API REST que se han creado, se ha seguido un mismo patrón MVC de tal forma que el modelo corresponderá a la representación de cada clase en formato JSON para los datos que se insertarán en la base de datos mongo DB. Las vistas son trasladadas bajo rutas hacia otra parte del proyecto ya sea entre servidores o entre servidor y aplicación y los controladores son las propias funciones que se realizaran bajo cada petición HTTP.

En cuanto al servidor de la aplicación, será necesario diseñar dos Api Rest para manejar tanto los usuarios que se registrarán en la app como los negocios que se inscribirán al funcionamiento de este proyecto de manera interna. Para ello y como se ha mencionado anteriormente, se dispondrá de una base de datos mongo DB donde se registrarán los distintos datos en formato JSON (ver apartado 2.6). Por tanto, el diseño conceptual de la base de datos será representado bajo los distintos modelos de los datos JSON. En este caso, los modelos son los siguientes:

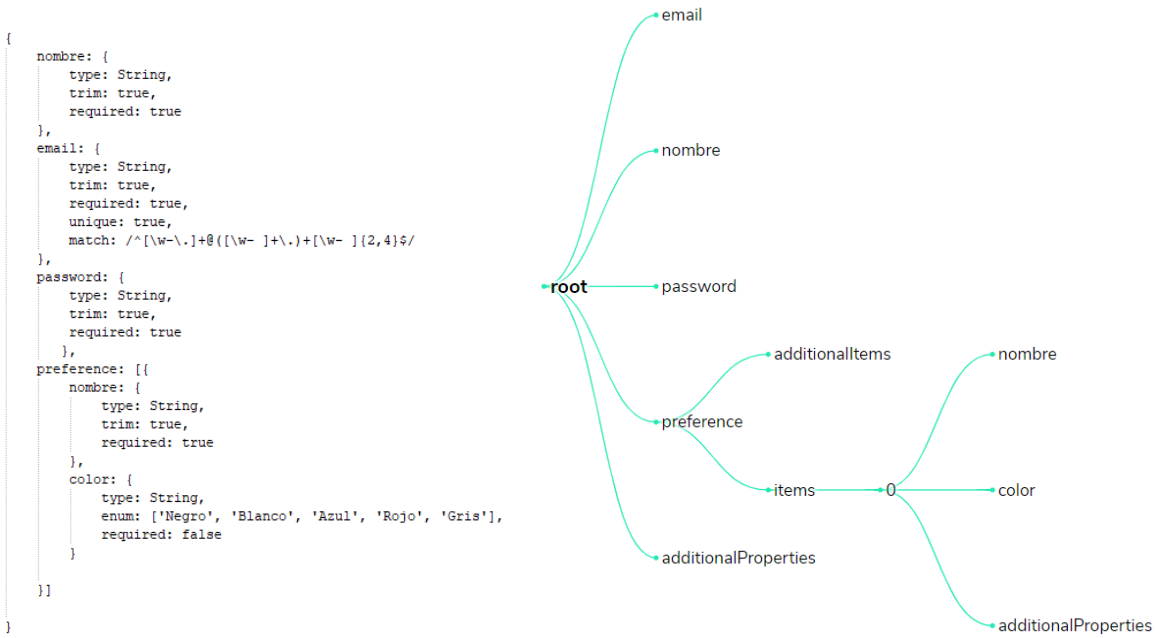


Figura 3.4 Diseño modelo de un usuario

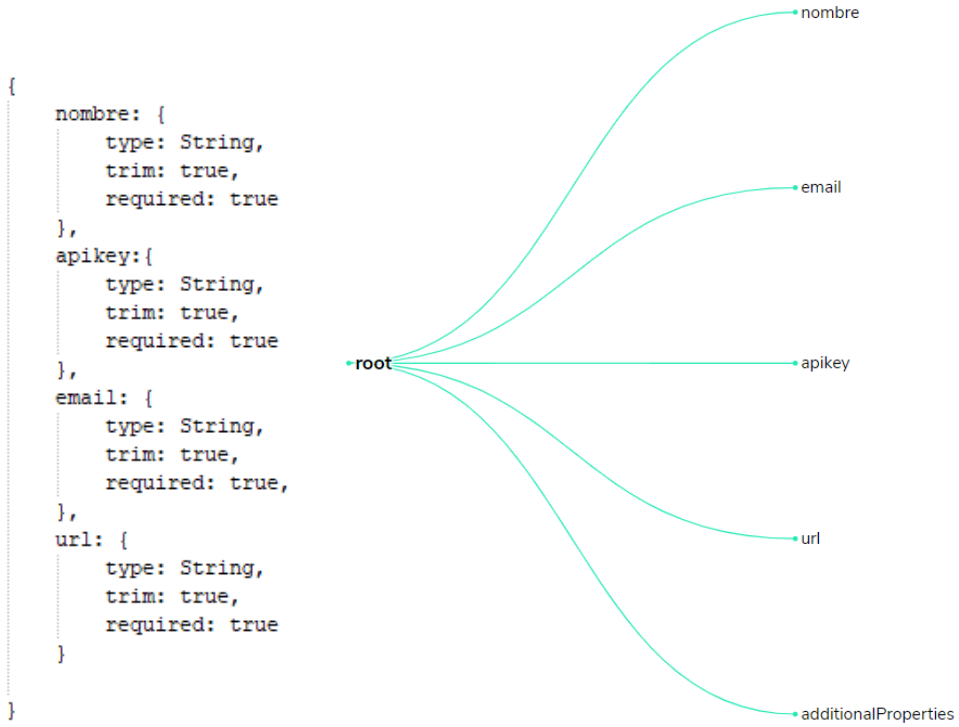


Figura 3.5 Diseño modelo de un negocio

Además, este servidor que denominaremos “central-server” dispondrá bajo su carpeta de recursos (src de source en inglés) los siguientes módulos implementados en Node.js:

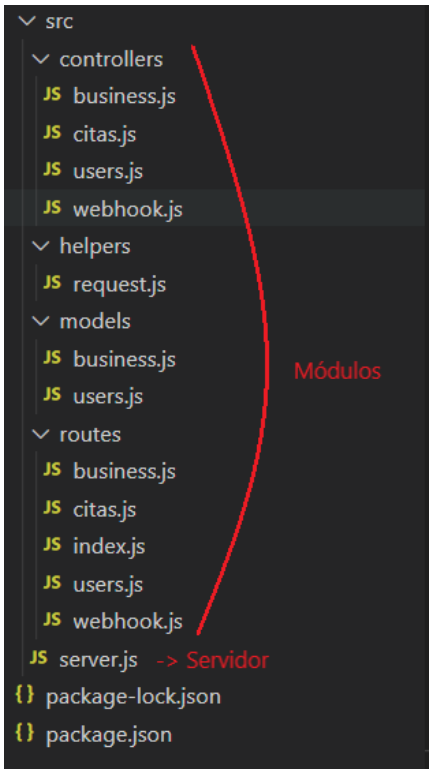


Figura 3.6 Jerarquía de archivos del servidor central de la aplicación

Referente al servidor del negocio y su lógica aplicada a la arquitectura, al ser una parte que debe añadir cada negocio individual que quiera participar en el proyecto y que solo se añadirá como prueba de concepto una versión para un negocio concreto (ver apartado 1.3), no se creará una base de datos mongo DB para registrar toda la información del negocio (puesto que esta debe estar alojada en el negocio). Para probar la lógica implementada, se crearán varios ficheros JSON con información relacionada con los productos disponibles en ese negocio y compras de clientes registrados en nuestra aplicación. Los datos de estos archivos siguen el siguiente modelo:

```
{
    "id": "2",
    "categoria": "Electronica",
    "tipo":  "Televisor",
    "producto": "Samsung TV QLED",
    "precio": 1000,
    "url": "https://images.samsung.com/is/image/samsung/cl-qledtv-q8cn-qn55q8cnaqxzs-frontttitanumsilver-102570240?$684_547_PNG$"
},
{
    "id": "3",
    "categoria": "Electronica",
    "tipo":  "Frigorifico",
    "producto": "Balay 3P",
    "precio": 60,
    "url": "https://www.electrodomesticosweb.es/images/product/1/large/pl_1_1_122431.png"
}
```

Figura 3.7 Diseño modelo de un producto

```
{
    "id": "1",
    "nombre": "Juanma",
    "email":  "Juanma@gmail.com",
    "compras": [{"producto": "Bosh Tassimo Happy",
                 "categoria": "Hogar",
                 "unidades": 3,
                 "precio": 10
                }],
    "cita": "2021/08/18 18:15:00"
},
{
    "id": "2",
    "nombre": "Andres",
    "email":  "Andres@gmail.com",
    "compras": [{"producto": "Samsung TV QLED",
                 "categoria": "Electronica",
                 "unidades": 2,
                 "precio": 1000
                }],
    "cita": "2018/01/30 23:30:14"
}
```


Figura 3.8 Diseño modelo de registro de compras y citas de clientes en el negocio

Además, este servidor que denominaremos “tfgjmmj” dispondrá bajo su carpeta de recursos (src de source en inglés) los siguientes módulos implementados en Node.js:

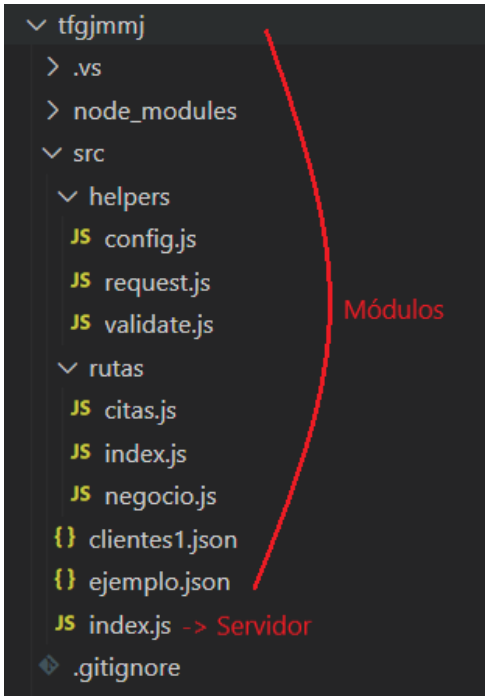


Figura 3.9 Jerarquía de archivos de la lógica y servidor de un negocio integrado a la app

3.2.3 Diagramas de secuencia

Para finalizar la fase de diseño se realizarán distintos diagramas de secuencia correspondientes a los funcionamientos elementales más característicos de este proyecto. La finalidad del diagrama de secuencia es poder visualizar el comportamiento dinámico de la arquitectura enfatizando en las distintas fases de intercambio de mensajes entre objetos del sistema.

Los diagramas diseñados son los siguientes:

- **Validar Acceso.** Como se puede visualizar en la figura 3.10, en el presente diagrama se describe el proceso de validación de acceso de un usuario que tiene instalada la aplicación. Si dicho usuario ha sido registrado previamente e inserta de manera correcta su email y su contraseña, será autorizado para acceder a la siguiente actividad y poder disfrutar de las distintas funcionalidades de la aplicación.

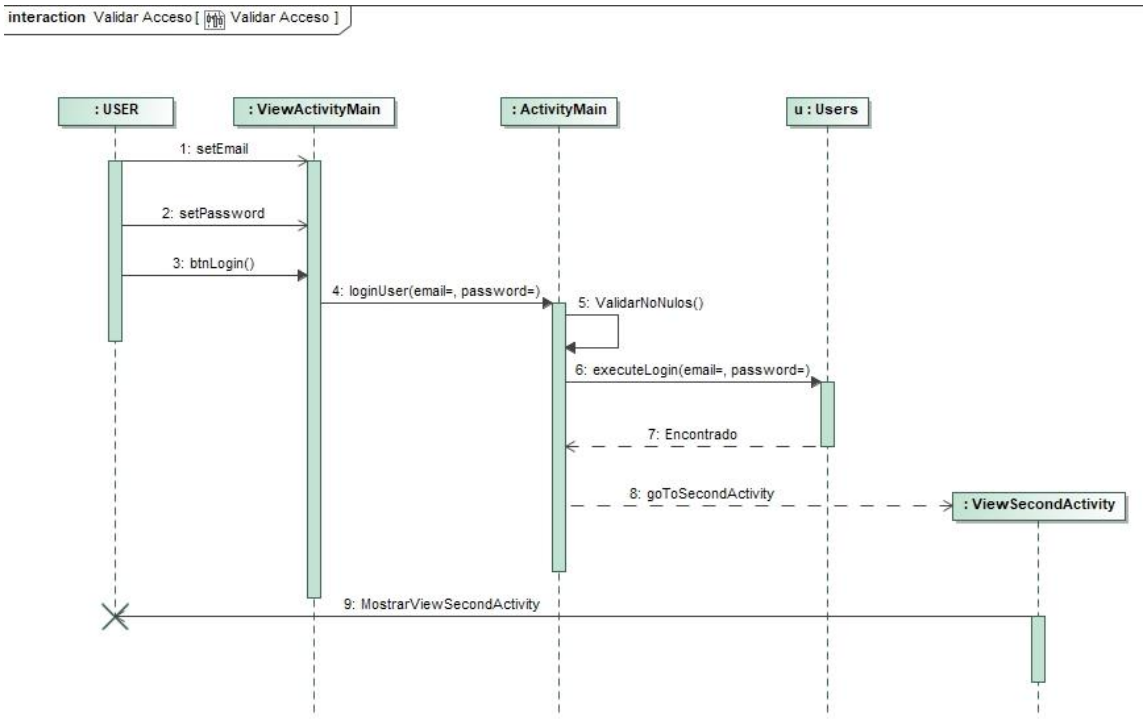


Figura 3.10 Diagrama de secuencia Validar Acceso

- **Registrar Usuario.** Como se aprecia en la figura 3.11, el diagrama representa la fase de registro de un nuevo usuario que instala la aplicación. Si el usuario completa de manera correcta el registro, insertando su nombre, email, contraseña y preferencias (estas últimas opcionales) será registrado de satisfactoriamente.

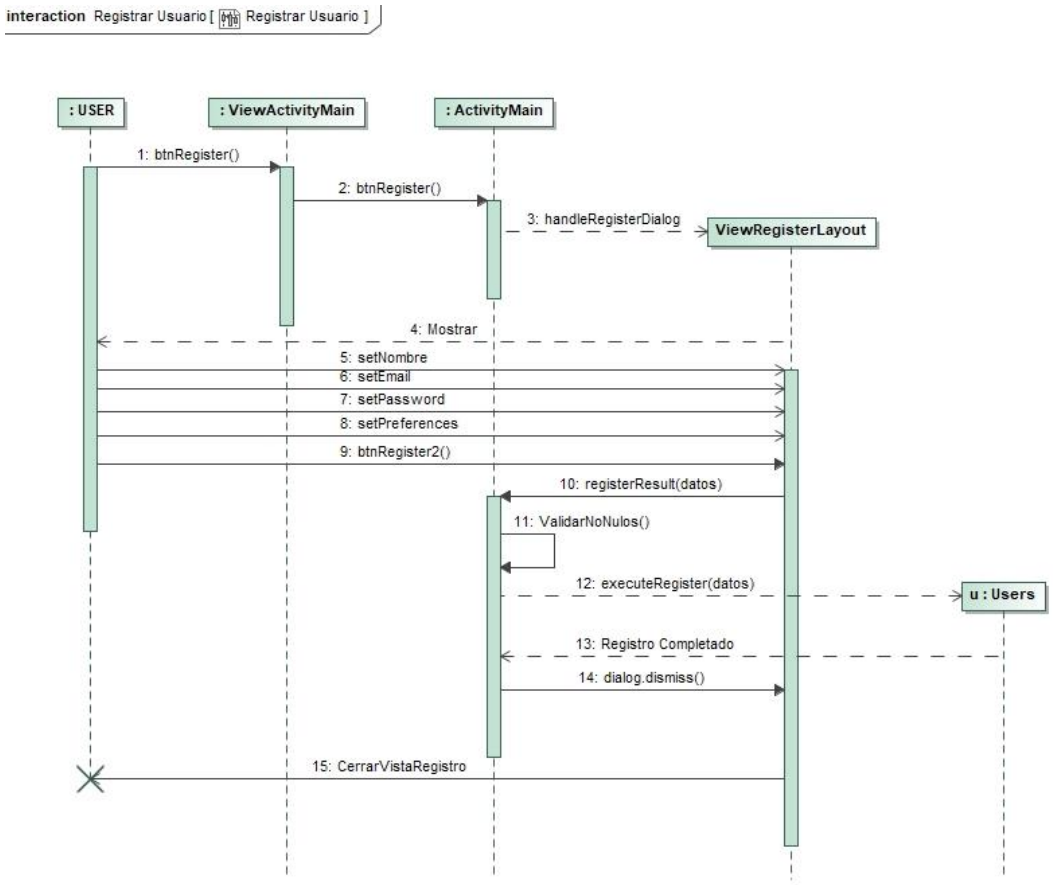


Figura 3.11 Diagrama de secuencia Registrar Usuario

- **Diagrama general de la arquitectura.** En el diagrama que se visualiza en la figura 3.12 se puede apreciar el proceso general que la arquitectura de este proyecto realiza para la obtención de beneficios. Si la aplicación detecta algún Beacon del Área Comercial y el usuario ha sido previamente validado en la aplicación, esta comenzará un proceso automático que enviará los datos del cliente al servidor del negocio del que se quiere obtener algún beneficio personalizado (ver apartado 2.4). A partir de la lógica de negocio que este mismo tenga implementada (pasos 11, 12 y 13) se obtendrá algún beneficio personalizado que se enviará al servidor de la aplicación y a través de Firebase a la aplicación en tiempo real (ver apartado 2.8).

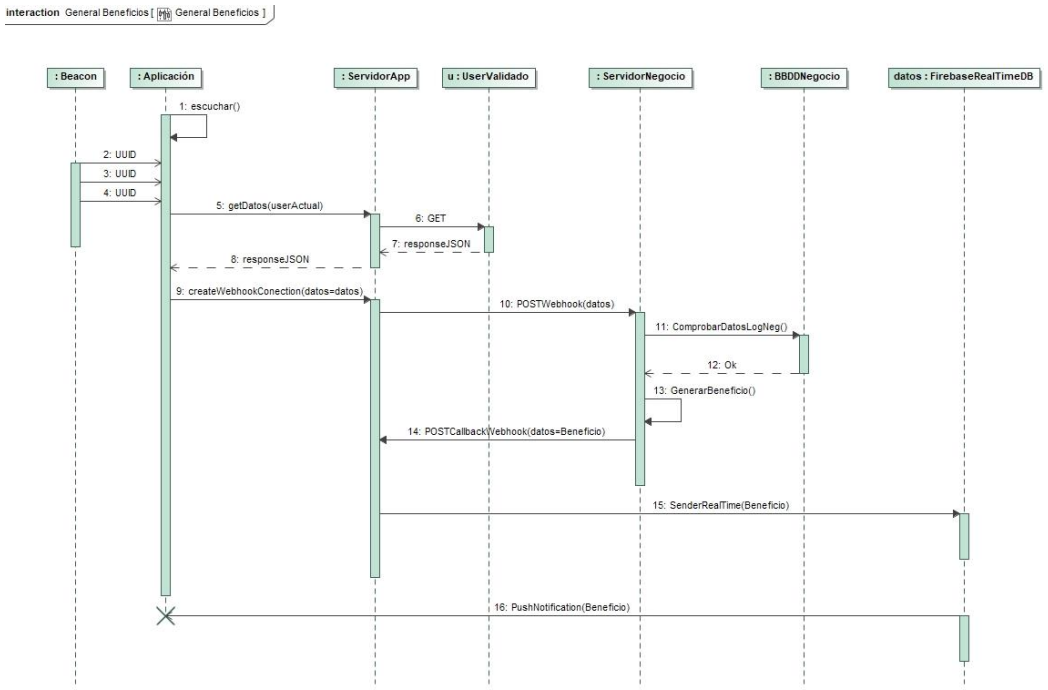


Figura 3.11 Diagrama de secuencia General de la Arquitectura

3.2.4 Interfaz común

Como se ha mencionado en los objetivos del proyecto (ver apartado 1.3), para poder integrar las distintas lógicas de negocio que cada comercio tendrá que implementar, es necesario definir un formato común de datos para que, independientemente de cómo se gestionen los datos en los servidores de los negocios, los beneficios que se elaboren deben seguir unas restricciones generales que denominaremos ***interfaz común.***

Las restricciones dependerán de los tipos de beneficios que se van a generar en este proyecto. En principio, las recompensas personalizadas que los clientes recibirán a través de la aplicación serán:

- Ofertas personalizadas de productos, según preferencias seleccionadas o compras realizadas.
- Recomendaciones de productos, según preferencias seleccionadas o una lista variada de elementos si no se ha indicado ninguna preferencia.
- Gestión personalizada de citas en negocios.

Cada beneficio que se genere dependerá de la situación personal de cada cliente, según sus compras realizadas en cada negocio, sus preferencias seleccionadas o sus citas disponibles. Por lo que las restricciones dependerán de estas situaciones para diferenciar entre un caso u otro. Debido a esto, la interfaz común será:

BENEFICIO	TIPO	DATOS	EXPLICACIÓN
Producto			
	OFERTA	{dato.tipo, dato.categoría, dato.producto, dato.precioAntes, dato.precioActual, dato.url, dato.negocio}	Oferta según preferencias
	RECOMENDACIÓN	{dato.tipo, dato.productos}	Recomendación según preferencias y compras
	OFERTACOMPRA	{dato.tipo, dato.categoría, dato.producto, dato.precioAntes, dato.precioActual, dato.url, dato.negocio}	Oferta según compras, sin preferencias
	RECOSINCOMPRA	{dato.tipo, dato.productos}	Recomendación sin preferencias ni compras
	RECOMENDACIÓN2	{dato.tipo, dato.productos}	Recomendación según preferencias, sin compras
	RECOSINCOMPRA2	{dato.tipo, dato.productos}	Recomendación sin preferencias, sin compras y sin registro en negocio
Cita			
	CAFÉ	{dato.tipo, dato.subtipo}	Asiste el día de la cita antes de su hora
	ESPERA	{dato.tipo, dato.subtipo}	Llega a su hora o en tiempo de cortesía
	PERDIDA	{dato.tipo, dato.subtipo}	Ha perdido su cita
	NOHOY	{dato.tipo, dato.subtipo, dato.fecha}	Su cita registrada no coincide en día con su asistencia
	NOEXISTE	{dato.tipo, dato.subtipo}	No tiene citas registradas

Figura 3.12 Tabla Interfaz Común de beneficios

4

Implementación y pruebas

Tras el diseño de los componentes del sistema, en esta fase se describe la implementación de todas las funcionalidades detalladas en las etapas anteriores. Posteriormente a la implementación, se documentarán las distintas pruebas de funcionamiento realizadas al sistema para corroborar su correcto funcionamiento.

Como en la fase anterior, para la implementación y las pruebas se seguirá desglosando la arquitectura del sistema en los distintos objetivos descritos anteriormente (ver apartado 1.3). De esta forma se aprecia de manera más sencilla cómo se ha ido desarrollando la arquitectura de manera incremental y cómo se diferencian y resuelven las distintas partes del sistema.

4.1 Implementación

4.1.1 Implementación del servidor de la aplicación

Para poder realizar acciones desde la aplicación web Android, es necesario disponer de un gestor que centralice y disponga todas las funcionalidades que esta pueda realizar. Por ello, será necesario, en principio, implementar un servidor para la aplicación.

El servidor será implementado en Node.js, tendrá módulos importados del propio lenguaje que facilitarán la creación del propio servidor, módulos que se han creado para implementar las funcionalidades requeridas por la aplicación, dos API REST que servirán de entorno para poder gestionar solicitudes respecto a los dos tipos de usuarios necesarios para la aplicación (clientes y negocios) y una base de datos mongo DB para almacenar los datos de usuarios.

- **Server.** Este será el fichero principal del servidor, en él se importan todos los módulos necesarios, los middlewares, las rutas, configuraciones, se conecta la base de datos y se inicia el propio dispositivo virtual.

```
const express = require('express');
const morgan = require('morgan');
const mongoose = require('mongoose');
const NotifServer = require('node-gcm');

const app = express();
```

Figura 4.1.1.1 Módulos utilizados de Node.js en server.js

En la figura 4.1.1.1 se puede visualizar los módulos de Node.js necesarios en el archivo server.js:

- Express es el módulo principal de Node.js que sirve para crear el servidor. A partir de él, se puede manejar peticiones HTTP, establecer ajustes de aplicaciones web como conexiones de puertos o localizaciones de plantillas y añadir procesamiento de peticiones middleware en cualquier punto dentro de la gestión de las distintas peticiones.
- Morgan es un middleware utilizado para registrar y realizar un seguimiento de los detalles de las solicitudes HTTP.
- Mongoose es el módulo que gestiona la base de datos mongo DB.
- Node-gcm se utilizará para enviar datos a la aplicación a través de notificaciones PUSH.

```
//DB
mongoose.connect('mongodb://localhost/central-server', { useNewUrlParser: true, useUnifiedTopology: true, useCreateIndex: true }
).then(db => console.log('db is connected')).catch(err => console.log(err));

//Settings
app.set('port', process.env.PORT || 4000);
app.set('json spaces', 2);

//middlewares
app.use(morgan('dev'));
app.use(express.urlencoded({extended: false}));
app.use(express.json());

//Routes
app.use(require('./routes/index'));
app.use('/api/business', require('./routes/business'));
app.use('/api/users', require('./routes/users'));
app.use('/app/citas', require('./routes/citas'));
app.use('/app', require('./routes/webhook'));
```

Figura 4.1.1.2 Continuación de fichero server.js

La figura 4.1.1.2 muestra la continuación del fichero server.js. En ella se puede visualizar la creación y conexión con la base de datos mongo DB que se conectará en la ruta indicada y si la ruta no existe la creará. Si la conexión se realiza de manera satisfactoria se mostrará por la consola el mensaje “db is connected” que indica que la base de datos está conectada.

Posteriormente se aprecia una sección de configuraciones (settings en inglés) donde se le indica al servidor que se conectará al puerto por defecto del sistema o al puerto 4000 si no hay ninguno establecido. También se indica que a los datos JSON se les añadan espacios para una mejor visualización.

A continuación, se usan los middlewares. Con Morgan indicamos que se quiere hacer un registro corto de datos con *'dev'*. *'express.urlencoded'* es utilizado para indicar al servidor que acepte datos recibidos de formularios (sin extensión para datos especiales como archivos de sonido, *'extended: false'*). Como última indicación para los middlewares del servidor, se utiliza express para poder recibir y enviar datos en formato JSON *'express.json()'*.

Por último, en la figura 4.1.1.2 se puede visualizar la sección de rutas. Estas son las utilizadas por el servidor para establecer los enlaces de conexión a las distintas API REST que se disponen y los puntos de entrada del propio servidor. Se puede apreciar que existen enlaces para la API REST de usuarios, para la API REST de los negocios y para establecer conexión con el servidor de los negocios a través de Webhook (ver apartado 2.4).

```
//Start the server
app.listen(app.get('port'), () => {
    console.log(`Server on port ${app.get('port')}`);
});
```

Figura 4.1.1.3 Inicio del servidor en server.js

En la figura 4.1.1.3 se muestra cómo se inicia el servidor en el puerto indicado anteriormente (ver figura 4.1.1.2).

Finalmente, en el fichero server.js, se dispone de dos entradas para solicitudes POST con relación a los distintos datos que los servidores de los negocios, a partir de su lógica implementada, podrán enviar al servidor de la aplicación. Los datos que serán enviados hacia estas entradas serán los referentes a ofertas personalizadas y recomendaciones de productos (a través de *'/webhook'*) y la gestión de citas de cada usuario clientes (a través de *'/citas'*). Si se reciben datos por estas entradas, estos serán enviados a la aplicación a través del envío de notificaciones PUSH de Firebase.

```
//Entrada webhook info negocios.
app.post('/webhook', function (req, res) {

    const sender = new NotifServer.Sender('AAAATVIsMEc:APA91bGmwFABwQIYIxDcaw2bMIhKtMNiJXr

    const message = new NotifServer.Message({
        notification: {
            title: "Has obtenido un beneficio",
            body: "Pulsa para obtener tu nuevo beneficio"
        },
        data: {
            datos: req.body
        }
    });

    const regTokens = [];
    regTokens.push("fC5OprZcS5aKKu4LiqqBMb:APA91bGFakkq0H8bHj-dCETjIODAU-qMzgDXp7qcmBjHKyI

    sender.send(message, { registrationTokens: regTokens}, function (err, response){
        if(err) console.error(err);
        else console.log(response);
    });

    res.json({message: "Dato recibido"});
```

Figura 4.1.1.4 Punto de entrada de solicitudes POST de beneficios

```
app.post('/citas', function (req, res) {

    const sender = new NotifServer.Sender('AAAATVIsMEc:APA91bGmwFABwQIYIxDcaw2bMIhKtMNiJX

    const message = new NotifServer.Message({
        notification: {
            title: "Se ha registrado su llegada",
            body: "Pulsa para ver el estado de su cita"
        },
        data: {
            datos: req.body
        }
    });

    const regTokens = [];
    regTokens.push("fC5OprZcS5aKKu4LiqqBMb:APA91bGFakkq0H8bHj-dCETjIODAU-qMzgDXp7qcmBjHKy

    sender.send(message, { registrationTokens: regTokens}, function (err, response){
        if(err) console.error(err);
        else console.log(response);
    });

    res.json({message: "Dato recibido"});
});
```

Figura 4.1.1.5 Punto de entrada de solicitudes POST de citas

- **Users.** Para implementar la API REST de los usuarios, como ya se ha mencionado anteriormente, se utilizará el patrón MVC (ver apartado 3.2.2). Para ello, se utilizarán 3 ficheros users.js; para el modelo, el controlador y la ruta.

En el modelo, se creará un nuevo esquema de mongo DB (en formato JSON) para poder almacenar a los usuarios con una forma determinada. El modelo de usuario será equivalente al mostrado en la etapa de diseño anterior (ver figura 3.4):

```
const userSchema = new Schema({
    nombre: {
        type: String,
        trim: true,
        required: true
    },
    email: {
        type: String,
        trim: true,
        required: true,
        unique: true,
        match: /^[\w-\.]+@([\w- ]+\.)+[\w- ]{2,4}$/
    },
    password: {
        type: String,
        trim: true,
        required: true
      },
    preference: [{
        nombre: {
            type: String,
            trim: true,
            required: true
        },
        color: {
            type: String,
            enum: ['Negro', 'Blanco', 'Azul', 'Rojo', 'Gris'],
            required: false
        }

    }]

});
```

Figura 4.1.1.6 Esquema modelo de un usuario

Las rutas que se utilizarán para acceder a las funcionalidades que la API REST de los usuarios dispone serán las siguientes:

```
const { Router } = require('express');
const router = Router();

const { getUsers, newUser, getOneUser, modifyUser, deleteUser, loginUser } = require('../controllers/users');

router.get('/', getUsers);
router.post('/register', newUser);
router.post('/login', loginUser)
router.get('/:email', getOneUser);
router.put('/edit/:email', modifyUser);
```

Figura 4.1.1.7 rutas API REST de usuarios

- La ruta *'api/users/'* sirve para obtener (petición GET) todos los usuarios disponibles en la base de datos.
- *'api/users/register'* será la ruta a la que se enviarán los datos de un nuevo usuario que se quiera registrar en la aplicación Android.
- Con *'api/users/login'* se realizará la validación de cuenta de un usuario que desee acceder a la aplicación.
- La ruta *'api/users/:email'* sirve para obtener los datos del usuario cuyo correo electrónico es enviado como parámetro en el propio enlace.
- Con *'api/users/edit/:email'* se pueden modificar las preferencias del usuario cuyo correo electrónico es enviado por el enlace.

Como se puede ver en la figura anterior, cada ruta guarda relación con una función que realiza la acción deseada en cada solicitud. Estas funciones pertenecen al controlador de usuarios. Las más importantes son:

```
newUser: async (req, res, next) => {

   const user = await Users.find({email: req.body.email});

    if(user.length >= 1){
            return res.status(409).json({ message: 'El email ya existe' });
        }else{
            try {
                const newUser = new Users(req.body);
                const user = await newUser.save();
                res.status(200).json(user);
            } catch (error) {
                console.log(error);
            }
    }
},
```

Figura 4.1.1.8 función para registrar usuarios

La figura anterior muestra la función necesaria para registrar un nuevo usuario a la aplicación. Primero se busca si el email enviado en el registro coincide con alguno ya registrado (solo se puede crear un usuario por correo electrónico, tomado como valor único); si no coincide, se crea el nuevo usuario, si coincide se genera un mensaje con estado HTTP 409 que dice *"El email ya existe"*.

```
    loginUser: async(req, res, next) => {

        const datos = req.body;
        const email = datos.email;
        const password = datos.password;

        const user = await Users.findOne({email: email});

        if(user == null){
                return res.status(409).json({ message: 'El email NO existe' });
            }else{
                try {

                    await Users.findOne({email: email}).then(function(us) {
                        return bcrypt.compare(password, us.password);
                    })
                    .then(function(samePassword) {
                        if(!samePassword){
                            res.status(403).send();
                        }else{
                        res.status(200).json({ message: 'Login success'});
                        }
                    })
                    .catch(function(err){
                        console.log("Error authenticating user...");
                        console.log(err);
                        next();
                    });

                } catch (error) {
                    console.log(error);
                }
        }

    }

};
```

Figura 4.1.1.9 función para validar la cuenta de un usuario

La figura 4.1.1.9 muestra la función que realiza la validación de cuenta de un usuario. Como se puede ver, la función comprueba que el email exista en la base de datos. Si existe, comprueba que la contraseña enviada y la que está registrada coincidan. Si finalmente coinciden, el usuario será autenticado de forma correcta.

```
modifyUser: async(req, res, next) => {

    const { email } = req.params;

    const user = await Users.findOne({email: email});

    if(user.length == 0){
        return res.status(404).json({ message: 'El usuario no existe' });
    }else{

        const newUser = user;
        newUser.preference = req.body;
        const oldUser =  await Users.findByIdAndUpdate(newUser._id, newUser);
        res.status(200).json({message: "Preferencias actualizadas"});
    }

},
```

Figura 4.1.1.10 función para validar la cuenta de un usuario

La figura anterior dispone la función necesaria para actualizar las preferencias del usuario cuyo correo electrónico se sitúa en *'req.params'*. Si el usuario existe en la base de datos, se modificarán las preferencias con los nuevos datos enviados en la solicitud por el propio usuario cliente.

- **Business.** De la misma forma que para la API REST de usuarios, para la API REST de los negocios inscritos a la aplicación, se utilizarán 3 ficheros business.js para representar el modelo, el controlador y las rutas establecidas.

El modelo implementado, equivalente al diseñado (ver figura 3.5), tiene el siguiente esquema:

```
const businessSchema = new Schema({
    nombre: {
        type: String,
        trim: true,
        required: true
    },
    apikey:{
        type: String,
        trim: true,
        required: true
    },
    email: {
        type: String,
        trim: true,
        required: true,
    },
    url: {
        type: String,
        trim: true,
        required: true
    }

});
```

Figura 4.1.1.11 Esquema modelo de un negocio

Los negocios no serán registrados desde la misma aplicación. Para poder inscribir un negocio a la arquitectura que este proyecto proporciona será necesario inscribirla de manera manual (esto podría ser una tarea para el gestor de la aplicación). Como se aprecia en la anterior figura, un negocio deberá aportar su nombre, una clave API para aportar seguridad al sistema, su correo electrónico de contacto y la URL por la que el servidor de la aplicación se conectará al servidor del propio negocio para obtener los datos necesarios.

Las rutas disponibles en la API REST de negocios mantienen una funcionalidad similar a la de usuarios. Las rutas son las siguientes:

```
const { Router } = require('express');
const router = Router();

const { getBusiness, newBusiness, getOneBusiness, modifyBusiness, deleteBusiness } = require('../controllers/business');

router.get('/', getBusiness);
router.post('/', newBusiness);
router.get('/:id', getOneBusiness);
router.put('/:id', modifyBusiness);
router.delete('/:id', deleteBusiness);

module.exports = router;
```

Figura 4.1.1.12 Rutas de la API REST de negocios

La función más importante y representativa para este sistema será la que permite registrar un nuevo negocio. Por ello, del controlador mostraremos dicha función denominada *"newBusiness"*.

```
newBusiness: async (req, res, next) => {

    const business = await Business.find({email: req.body.email});

    if(business.length >= 1){
            return res.status(409).json({ message: 'El negocio ya existe' });
        }else{
            try {
                const newBusiness = new Business(req.body);
                const business = await newBusiness.save();
                res.status(200).json(business);
            } catch (error) {
                console.log(error);
            }
    }
},
```

Figura 4.1.1.13 Función para inscribir un nuevo negocio al sistema

- **Request.** El archivo request.js se ha creado para poder reutilizar una función que se va a utilizar múltiples veces en este sistema. Gracias al módulo de Node.js *'request-promise'* y la función creada se podrán realizar solicitudes HTTP desde cualquier punto de este servidor.

```
const rp = require('request-promise');
const prettyjson = require('prettyjson');

const apiRest = 'http://localhost:3000/';
const prettyjsonOptions = {};

function request(path, method,apikey, body) {
    let options = {
        url: `${apiRest}${path}`,
        method: method,
        json: true,
        headers: {
            'secretkey': apikey
        }

    };

    if (body)
        options.body = body;
    console.log(prettyjson.render(options, prettyjsonOptions));
    return rp(options);
}

module.exports = {
    request: request
};
```

Figura 4.1.1.14 request.js

- **Webhook.** Los ficheros webhook.js de este servidor se utilizarán para crear e iniciar el proceso automático de obtención de beneficios realizado mediante Webhook (ver apartado 2.4).

```
const { Router } = require('express');
const requests = require('../helpers/request');
const router = Router();

const { createWebHookConnection } = require('../controllers/webhook');

router.post('/webhook', createWebHookConnection);


module.exports = router;
```

Figura 4.1.1.15 Ruta webhook.js

```
createWebHookConnection: async (req,res, next) => {

    const usuario = await UserM.findOne({email: req.body.email});
    const negocio = await Business.findOne({email: businessEmail});

    const negocioApiKey = negocio.apikey;

    if(usuario != null){


        let body = {};

        body.nombre = usuario.nombre;
        body.email = usuario.email;
        body.preference = req.body.preference.slice();

        res.status(200).json({ message: 'Webhook creado correctamente'});

        return requests.request(`api/${businnesName}/webhook`, 'POST', negocioApiKey, body);
    }else{
        return json({ message: 'El usuario no existe' });
    }

}
```

Figura 4.1.1.16 función que inicia el webhook para obtener beneficios

La figura 4.1.1.16 muestra la función que crea la conexión Webhook. Tras comprobar si los datos aportados del usuario y negocio son válidos se crea un conjunto de datos JSON denominado *"body"* , en él se guardan los datos del usuario cliente y se envían al servidor del propio negocio (solicitud POST), junto a la clave api del negocio en cuestión y a la URL que se genera al añadir a la indicada en request.js (ver figura 4.1.1.14) y la que se indica en los parámetros de entrada de la función *request* de esta figura.

- **Citas.** De manera similar a Webhook (punto anterior), en los ficheros citas.js se iniciará una conexión Webhook para poder gestionar las citas que el usuario cliente pueda disponer en los negocios registrados en la aplicación.

```
const { Router } = require('express');
const requests = require('../helpers/request');
const router = Router();

const { setLlegadaCita } = require('../controllers/citas');

router.post('/', setLlegadaCita);


module.exports = router;
```

Figura 4.1.1.17 Ruta citas.js

```
setLlegadaCita: async (req,res, next) => {

    const usuario = await UserM.findOne({email: req.body.email});
    const negocio = await Business.findOne({email: businessEmail});

    const negocioApiKey = negocio.apikey;


    if(usuario != null){


        let body = {};

        body.email = usuario.email;
        body.llegada = req.body.llegada;

        res.status(200).json({ message: 'Llegada registrada correctamente ...'});

        return requests.request(`api/${businnesName}/citas`, 'POST', negocioApiKey, body);
    }else{
        return json({ message: 'El usuario no existe' });
    }

}
```

Figura 4.1.1.18 función que inicia el Webhook para gestionar citas

En la figura anterior se puede visualizar la función que inicia el proceso automático que gestionará las posibles citas que un usuario pueda obtener en los distintos negocios que están registrados al sistema. Tras comprobar que el usuario y negocio son válidos, se creará un conjunto de datos denominado *"body"* en el que se guardarán el correo electrónico del cliente y su hora de llegada a la cita. Estos datos se enviarán mediante la función de *request* (con una petición POST) al punto de entrada del servidor del negocio que gestione las citas de los clientes.

4.1.2 Implementación del servidor de negocio

Una vez se ha iniciado el proceso automático de Webhook para la obtención de beneficios personalizados, es necesario crear el servidor que recibe dicha información e implementar la lógica de negocio necesaria para gestionar los datos recibidos, realizar comprobaciones respecto a la información disponible en la base de datos del negocio en cuestión y elaborar el servicio personalizado que se enviará de vuelta al servidor de la app y posteriormente a la aplicación.

Cabe destacar que cada negocio que quisiera inscribirse al sistema debería implementar su propio servidor, crear su lógica de negocio e integrarlos al sistema bajo las restricciones que este impone. Para obtener una visión completa de la infraestructura y poder obtener resultados finales, se implementará como prueba de concepto una versión de este componente para un negocio concreto, añadiendo tanto la lógica implementada para el negocio (acorde a las restricciones de la aplicación) y el servidor necesario para ese negocio.

- **Index.** Fichero similar a *server.js* del punto anterior (ver apartado 4.1.1), en este archivo se registran los módulos Node.js, las funciones que procesan datos antes que el servidor los reciba (middlewares), las configuraciones necesarias, las rutas que son accesibles y reciben solicitudes HTTP; y se inicia el servidor, en este caso, en el puerto en el puerto definido por el sistema o el 3000.

```
const express = require('express');
const app = express();
const morgan = require('morgan'); //It allows to see by console what will arrive at the server.



/* Settings */
app.set('port', process.env.PORT || 3000); //To obtain from any part of the app, PORT in case there is a defined port in the system (cloud case).
app.set('json spaces', 2);

/* Middlewares: Function that processes data before the server receives it */
app.use(morgan('dev'));
app.use(express.json()); //Allows the server to receive and understand JSON formats.
app.use(express.urlencoded({extended: false})); //To understand data received from forms, false for simple data (only input not img).

/* Routes */
app.use(require('./rutas/index'));
app.use('/api/negocio', require('./rutas/negocio'));
app.use('/api/negocio/citas', require('./rutas/citas'));


/* Starting server */
app.listen(app.get('port'), () => {
    console.log(`Server on port ${app.get('port')}`);
});
```

Figura 4.1.2.1 Fichero index.js del servidor de negocio

Como podemos ver en las rutas disponibles, existen dos accesos claves que permitirán al servidor poder ofrecer las funcionalidades de gestión de beneficios personalizados. A la ruta *'/api/negocio'* se accederá para analizar las compras de los clientes que desean obtener recompensas y poder elaborar ofertas o recomendaciones de productos según la situación de cada usuario cliente. Para poder gestionar las citas que un cliente pudiese tener en el negocio en cuestión, será necesario enviar solicitudes bajo la ruta *'/api/negocio/citas'*.

- **Config.** El fichero config.js es el encargado de guardar la clave API que este negocio tiene implementada como método de seguridad. Esta clave podrá ser leída desde el propio sistema si se ha ocultado bajo *dotenv* (proceso de enmascaramiento de claves en servidores) o la clave disponible por defecto.

```
module.exports = {
    SECRET_KEY: process.env.SECRET_KEY || 'sQ:%9O@SK3PJw7cCKXmE',
};
```

Figura 4.1.2.2 Fichero config.js del servidor de negocio

- **Validate.** Para poder validar el acceso a las funcionalidades que el servidor del negocio posee, es necesario crear una función que ofrezca ese servicio de autorización. En el fichero validate.js se ha implementado la función *validateKey* que realiza el proceso anteriormente detallado.

```
const config = require('./config');

const validateKey = (req, res, next) => {

    if (!req.headers.secretkey) {
        return res
          .status(403)
          .send({ message: "Tu petición no tiene cabecera de clave secreta" });
    }

    const apikey = req.headers.secretkey;
    const ak_original = config.SECRET_KEY;

    if(apikey == ak_original){

        console.log("Clave API Correcta ...");
        next();

    }else{
        return res
        .status(403)
        .send({ message: "La clave API no es correcta" });
    }

};

  module.exports = {validateKey};
```

Figura 4.1.2.3 Fichero validate.js del servidor de negocio

Como se puede apreciar en la figura 4.1.2.3, de la cabecera de la petición se extrae la clave api secreta que se ha enviado desde el cliente. Si existe la clave, se compara con la original y se permite el acceso si coinciden o se deniega en el caso opuesto.

- **Negocio.** En este fichero reside la lógica de negocio en cuanto a elaboración de ofertas y recomendaciones personalizadas de productos. En negocio.js está el punto de entrada de las peticiones que se realizarán desde el servidor de la aplicación con los datos del cliente que desee obtener beneficios personalizados (a partir del proceso automático de Webhook). Una vez se ha recibido esos datos, se gestionarán en una función que sirve de manejador de la información, se generará un determinado beneficio según las restricciones del sistema y las circunstancias del cliente y finalmente se enviará dicho beneficio mediante una solicitud POST hacia el servidor de la aplicación, terminando así, el proceso de Webhook.

```
router.post('/webhook', validate.validateKey, function (req, res) {

    const apikey = req.headers.secretkey;

    if (req.body) {
        handleRequest(req, res);
    } else {
        res.status(401).send("Forbidden");
    }
});
```

Figura 4.1.2.4 Endpoint del servidor de negocio

Si se visualiza la figura anterior, se puede apreciar el punto de entrada de las solicitudes POST por parte del servidor con relación a las ofertas y recomendaciones de productos. Puede verse como, antes de proceder a la comprobación de la existencia de los datos y su manejo, es necesario autorizar el acceso a partir de la validación de la clave API.

Si *"validateKey"* corrobora que la clave API es correcta, se procederá, si existen datos en el cuerpo de la solicitud, al manejo de la información.

Para gestionar la información, en esta lógica de negocio concreta, se ha decidido implementar funciones auxiliares que facilitan la labor de búsqueda en la base de datos del propio establecimiento. Las funciones son:

```
function buscarCliente(email){

    let sol;

    _.each (clientes, (cl, i) => {
        if(cl.email === email) {

            sol = cl;
        }
    });

    return sol;
}
```

Figura 4.1.2.5 Función buscar cliente

La función *"buscarCliente"* busca a partir del email del cliente que envía los datos, si existe información en la base de datos del negocio respecto a ese usuario. Si existe información devuelve su valor, sino devolverá *"undefined"* (valor por defecto de las variables en Javascript).

```
function buscarProductoRel(cat, prodct){

    let sol;

    _.each (negocio, (prod, i) => {
        if(prod.categoria === cat && prod.producto !== prodct ) {
            sol = prod;
        }
    });

    return sol;

}
```

Figura 4.1.2.6 Función buscar producto relacionado

La función *"buscarProductoRel"* busca un producto relacionado a una categoría y producto dados. Si encuentra algún producto que pertenezca a la misma categoría que el producto enviado y no es ese mismo producto, lo devolverá como solución. Esta función es útil para buscar productos que se relacionen con las preferencias que el usuario registra en su aplicación o para buscar productos relacionados a las compras de un usuario.

```
function buscarProductosRel(prefe, pdts){


    _.each (negocio, (prod, i) => {
        if(prod.categoria === prefe && pdts.length < 5 ) {
            pdts.push(prod);
        }
    });

    return pdts;

}
```

Figura 4.1.2.7 Función buscar productos relacionados

La función de la figura anterior es similar a la función de la figura 4.1.2.6. La diferencia está en la cantidad de productos relacionados que devuelve cada función. *"buscarProductosRel"* generará una lista de productos relacionados de tamaño entre 0 y 5, es decir, esta función buscará productos relacionados en la base de datos del negocio generando una lista de máximo 5 productos. Esta función se utilizará para generar recomendaciones de productos para usuarios que no han realizado compras en el negocio.

Una vez definidas las funciones auxiliares que se utilizarán, será necesario describir con detalle, el manejador que gestiona la lógica de este punto del sistema.

Teniendo en cuenta las restricciones que la aplicación impone referente al modelo de los beneficios generados (ver apartado 3.2.4), se ha de implementar el manejador de tal forma que se abarquen todas las situaciones posibles. Primeramente, en el manejador se comprobará si el cliente que desea obtener beneficios está registrado en el negocio (ha realizado compras). Si no ha realizado compras, será necesario verificar si el usuario ha seleccionado preferencias que coincidan con categorías de productos disponibles en este negocio en concreto. Si estas preferencias existen, se generará una lista de productos seleccionados acorde a sus prioridades. Por el contrario, si no ha seleccionado preferencias se elaborará una lista de productos con categorías al azar.

```
function handleRequest(req, res) {

    let event = req.body;

    let cliente = buscarCliente(event.email);
    let preference = event.preference;



    let i = 0;
    var encontrado = false;

    if(cliente !== undefined){ //Existe el cliente en el negocio

        let compras = cliente.compras;
```

Figura 4.1.2.8 Primeras comprobaciones del manejador

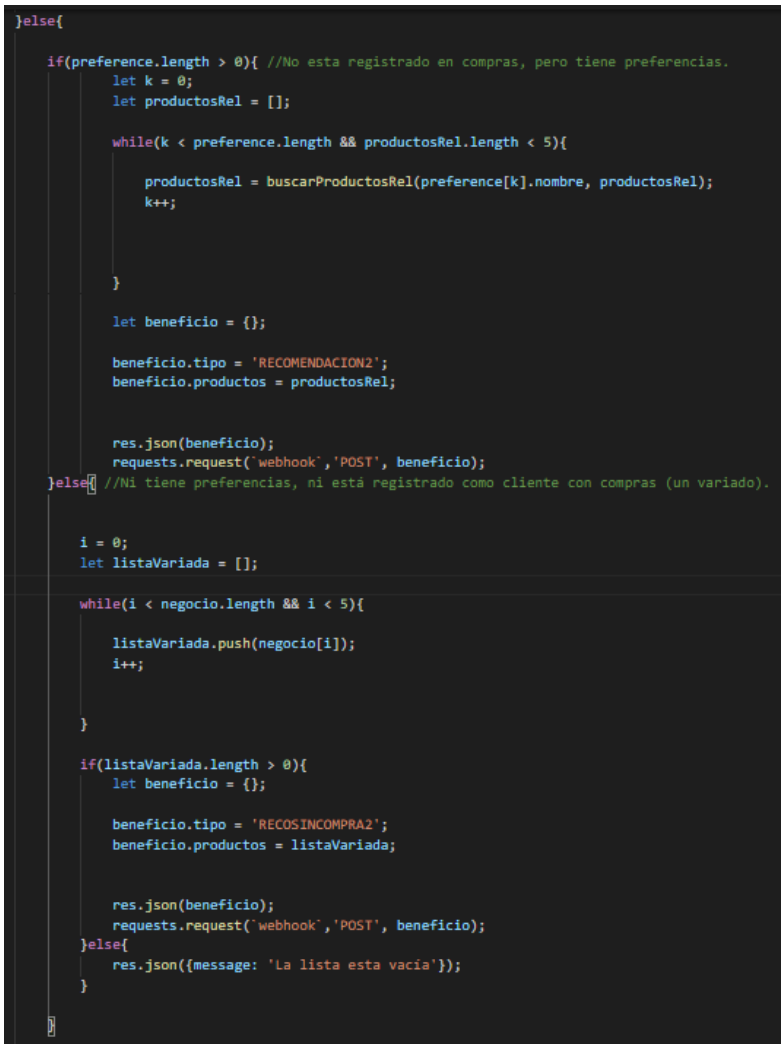


Figura 4.1.2.9 Primeras comprobaciones del manejador 2

Dada la primera opción, es decir, si el cliente tiene compras registradas en el negocio, el segundo paso será comprobar si dicho usuario ha registrado preferencias en su aplicación. En caso afirmativo, se comprobará si las preferencias coinciden con la categoría de algún producto de los que haya comprado con anterioridad. En caso negativo, si tiene compras registradas se realizará una oferta personalizada de algún producto al azar que coincida con la categoría de algún producto que haya comprado; si no tiene compras registradas, se generará una lista de recomendaciones de productos con categorías al azar.

Figura 4.1.2.10 Caso afirmativo.

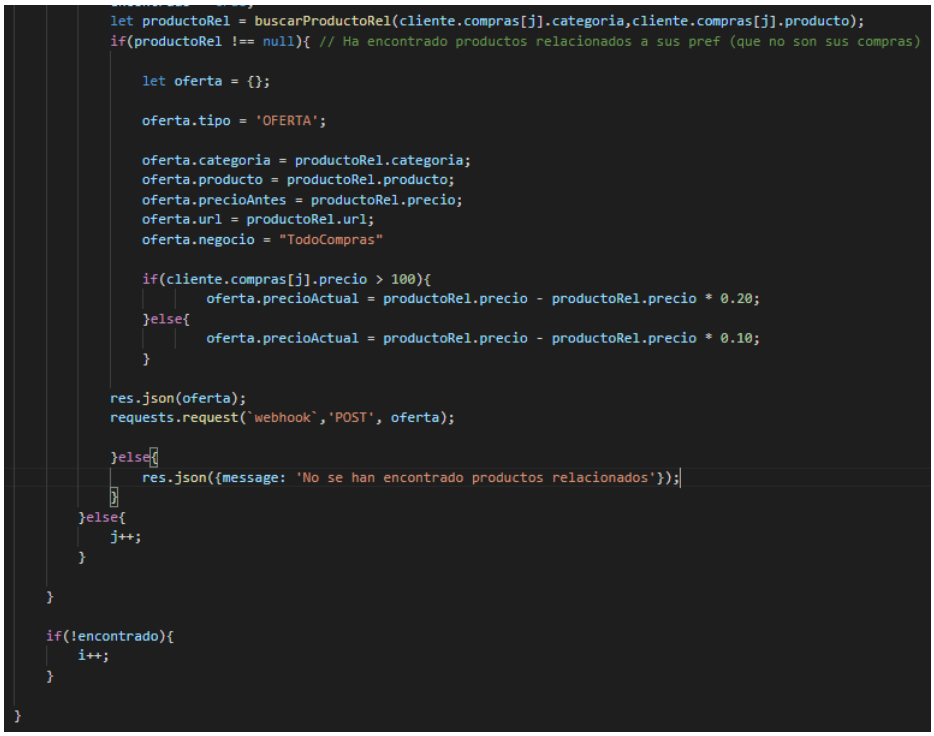
```
}else{ //Se ha registrado sin preferencias



    i = 0;
    let ofertaCompra = []

    if(compras.length > 0){ //Si ha comprado, OFERTA según sus compras.

        compras.forEach((pdt) => {

            let product = buscarProductoRel(pdt.categoria, pdt.producto);

            if(product !== null){

                ofertaCompra.push(product);

            }


        });

        if(ofertaCompra.length > 0){



            let random = generateRandomInt(ofertaCompra.length-1);

            let oferta = {};

            oferta.tipo = 'OFERTACOMPRA';

            oferta.categoria = ofertaCompra[random].categoria;
            oferta.producto = ofertaCompra[random].producto;
            oferta.precioAntes = ofertaCompra[random].precio;
            oferta.precioActual = ofertaCompra[random].precio - ofertaCompra[random].precio * 0.20;
            oferta.url = ofertaCompra[random].url;
            oferta.negocio = "TodoCompras";


            res.json(oferta);
            requests.request(`webhook`,'POST', oferta);


        }
```

Figura 4.1.2.11 Caso negativo, primera opción

```
    }else{ //Si no ha comprado, un variado.

        i = 0;
        let listaVariada = [];

        while(i < negocio.length && i < 5){

            listaVariada.push(negocio[i]);
            i++;


        }

        if(listaVariada.length > 0){
            let beneficio = {};

            beneficio.tipo = 'RECOSINCOMPRA';
            beneficio.productos = listaVariada;


            res.json(beneficio);
            requests.request(`webhook`,'POST', beneficio);
        }else{
            res.json({messaje: 'La lista esta vacia'});
        }

    }
```

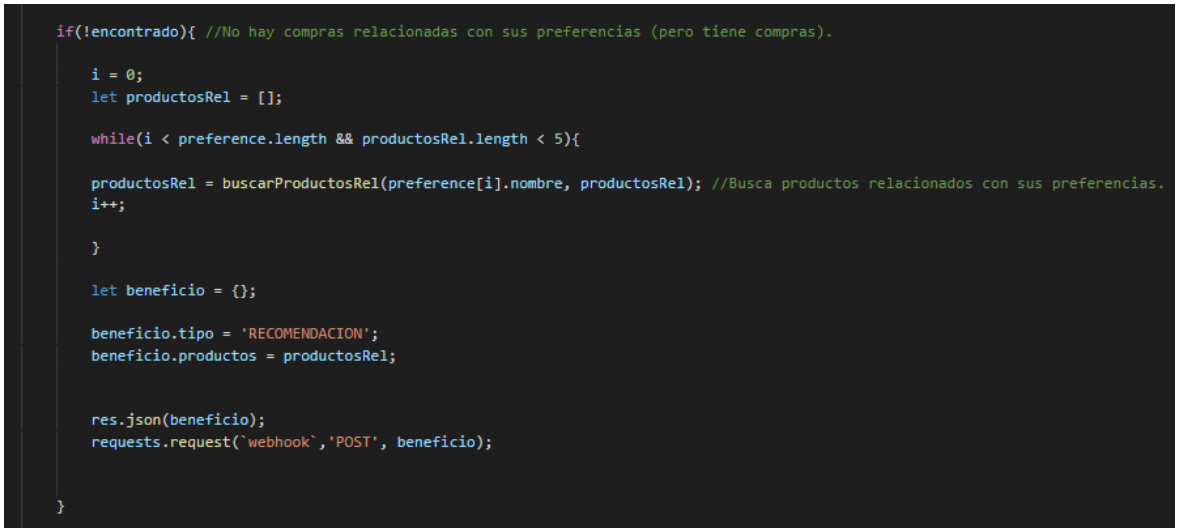
Figura 4.1.2.12 Caso negativo, segunda opción

Si continúa el procedimiento por el caso afirmativo anteriormente mencionado (ver figura 4.1.2.10), es decir, si se ha encontrado productos comprados que coinciden en categoría con sus preferencias, se deberá buscar productos relacionados con dicha preferencia y el producto que posee su misma categoría encontrado. En supuesto caso de encontrar productos relacionados, se generará una oferta con algún producto relacionado. Esta oferta dependerá del precio del producto que haya comprado anteriormente, si dicho producto comprado supera un valor determinado, el porcentaje de oferta será mayor, por el contrario, en caso de no superarlo, la oferta será menor. En el caso de no encontrar productos relacionados, se generará una lista de productos recomendados según sus compras.



```
            let productoRel = buscarProductoRel(cliente.compras[j].categoria,cliente.compras[j].producto);
            if(productoRel !== null){ // Ha encontrado productos relacionados a sus pref (que no son sus compras)

                let oferta = {};

                oferta.tipo = 'OFERTA';

                oferta.categoria = productoRel.categoria;
                oferta.producto = productoRel.producto;
                oferta.precioAntes = productoRel.precio;
                oferta.url = productoRel.url;
                oferta.negocio = "TodoCompras"

                if(cliente.compras[j].precio > 100){
                        oferta.precioActual = productoRel.precio - productoRel.precio * 0.20;
                }else{
                        oferta.precioActual = productoRel.precio - productoRel.precio * 0.10;
                }

            res.json(oferta);
            requests.request(`webhook`,'POST', oferta);

            }else{
                res.json({message: 'No se han encontrado productos relacionados'});
            }
        }else{
            j++;
        }

    }

    if(!encontrado){
        i++;
    }
}
```

Figura 4.1.2.13 Caso afirmativo, genera oferta



```
if(!encontrado){ //No hay compras relacionadas con sus preferencias (pero tiene compras).

    i = 0;
    let productosRel = [];

    while(i < preference.length && productosRel.length < 5){

    productosRel = buscarProductosRel(preference[i].nombre, productosRel); //Busca productos relacionados con sus preferencias.
    i++;

    }

    let beneficio = {};

    beneficio.tipo = 'RECOMENDACION';
    beneficio.productos = productosRel;


    res.json(beneficio);
    requests.request(`webhook`,'POST', beneficio);


}
```

Figura 4.1.2.14 Caso negativo, genera recomendación

- **Citas.** El fichero citas.js dispone de un proceso similar al punto anterior, pero para la gestión de citas. Mediante un punto de entrada de solicitudes POST, el servidor de la aplicación enviará al servidor de negocio tanto la información de usuario (su correo electrónico) como la fecha exacta a la que indica que está esperando su cita. Tras recibir los datos mencionados, a partir de un manejador, se tramitará la información para elaborar una respuesta según los datos sobre las citas de los usuarios clientes que se disponga en la base de datos del negocio en concreto. Una vez se elabore la respuesta, esta será enviada al servidor de la aplicación a través de una solicitud POST que, a su vez, finalizará el proceso automático de Webhook.

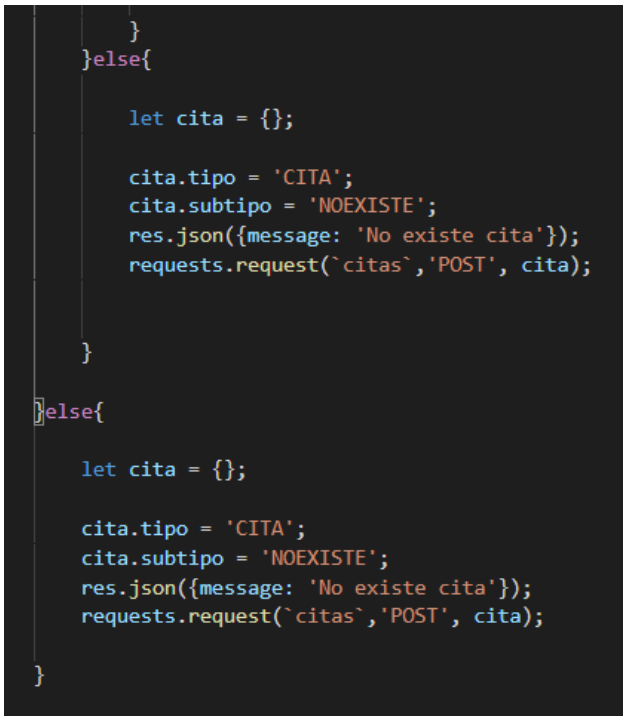
```
router.post('/', validate.validateKey, function (req, res) {

    if (req.body) {
        handleRequest(req, res);
    } else {
        res.status(401).send("Forbidden");
    }
});
```

Figura 4.1.2.15 Endpoint de citas en citas.js

El gestor de citas comenzará comprobando la información que recibe desde el servidor de la aplicación. Se iniciará una búsqueda del cliente que desea gestionar su cita. Si este cliente no se encuentra en la base de datos, evidentemente, no tendrá ninguna cita que gestionar y se informará al usuario de que no existe ninguna cita para él. Por el contrario, si el cliente se encuentra en la base de datos, se verificará si en su registro de citas posee algún dato. Si no hay ninguna cita disponible, se informará al cliente de ello.

```
function handleRequest(req, res) {

    let event = req.body;

    let cliente = buscarCliente(event.email);
    let llegada = event.llegada;

    if(cliente !== undefined){ //Existe el cliente en el registro del negocio

        let citaDato = cliente.cita;

        if(citaDato !== null){ //Tiene cita
            const dateLlegada = new Date(llegada);
            const dateCita = new Date(citaDato);
```

Figura 4.1.2.16 Comprobación de información en manejador de citas

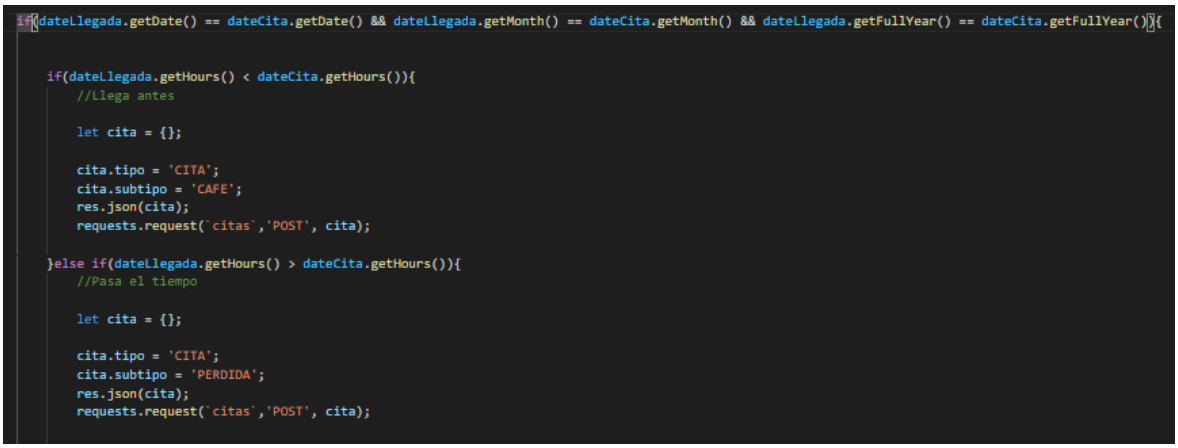


```
        }
    }else{

        let cita = {};

        cita.tipo = 'CITA';
        cita.subtipo = 'NOEXISTE';
        res.json({message: 'No existe cita'});
        requests.request(`citas`,'POST', cita);


    }

}else{

    let cita = {};

    cita.tipo = 'CITA';
    cita.subtipo = 'NOEXISTE';
    res.json({message: 'No existe cita'});
    requests.request(`citas`,'POST', cita);

}
```

Figura 4.1.2.17 Envío de información sobre la no existencia de citas

En el caso de tener citas registradas, será necesario comparar la fecha en la que el usuario indica que está esperando su cita y la fecha de la cita en cuestión. Cabe destacar que en Node.js y Javascript existen funciones predeterminadas para extraer de un dato de fecha (tipo Date) la información que se requiera. Por ello, será necesario extraer de las dos fechas cada dato disponible (día, mes, año y hora) para poder realizar una comparación exhaustiva.

En el caso de asistir a la cita un día que no es el de la cita disponible, se informará al usuario de que su reunión no es ese mismo día (adjuntando el día real de su cita). Si asiste en su día y llega antes de su hora de cita, el sistema generará un beneficio para cliente (ofrecerá un café gratis mientras espera su turno). En el caso de llegar justo a su hora o en un intervalo de retraso de 10 minutos (tiempo de espera de cortesía seleccionado en esta lógica de negocio) se informará al cliente de que se le está esperando en un lugar específico del Área Comercial. En el caso de sobrepasar el tiempo de espera de cortesía seleccionado, se informará al usuario de que su cita ha expirado.



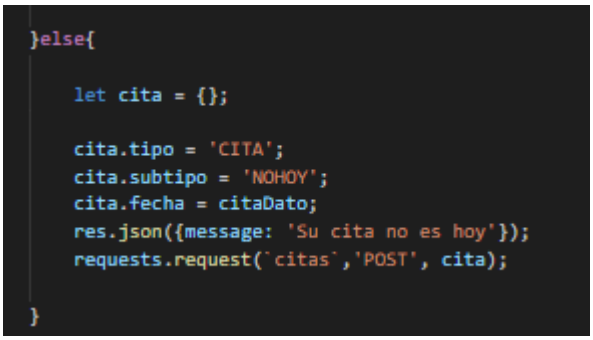
```
if(dateLlegada.getDate() == dateCita.getDate() && dateLlegada.getMonth() == dateCita.getMonth() && dateLlegada.getFullYear() == dateCita.getFullYear()){


    if(dateLlegada.getHours() < dateCita.getHours()){
        //Llega antes

        let cita = {};

        cita.tipo = 'CITA';
        cita.subtipo = 'CAFE';
        res.json(cita);
        requests.request(`citas`,'POST', cita);

    }else if(dateLlegada.getHours() > dateCita.getHours()){
        //Pasa el tiempo

        let cita = {};

        cita.tipo = 'CITA';
        cita.subtipo = 'PERDIDA';
        res.json(cita);
        requests.request(`citas`,'POST', cita);
```

Figura 4.1.2.18 Gestión de citas 1

```
}else{
    //comparar minutos
    if(dateLlegada.getMinutes() <= dateCita.getMinutes()){

        //Llega antes

        let cita = {};

        cita.tipo = 'CITA';
        cita.subtipo = 'CAFE';
        res.json(cita);
        requests.request(`citas`,'POST', cita);

    }else{
        //llega tarde pero revisar

        if(dateLlegada.getTime()-dateCita.getTime() <= 600000){
            //Llega a tiempo

            let cita = {};

            cita.tipo = 'CITA';
            cita.subtipo = 'ESPERA';
            res.json(cita);
            requests.request(`citas`,'POST', cita);


        }else{
            //Llega tarde

            let cita = {};

            cita.tipo = 'CITA';
            cita.subtipo = 'PERDIDA';
            res.json(cita);
            requests.request(`citas`,'POST', cita);
        }

    }
}
```

Figura 4.1.2.19 Gestión de citas 2

De la figura anterior cabe destacar el caso de llegar en un intervalo de tiempo entre la hora de la cita y 10 minutos más tarde. Para comparar ese periodo de tiempo se ha restado la hora de la llega con la hora de la cita y se ha comprobado que esta sea menor o igual a 600000 milisegundos (10 minutos).



```
}else{

    let cita = {};

    cita.tipo = 'CITA';
    cita.subtipo = 'NOHOY';
    cita.fecha = citaDato;
    res.json({message: 'Su cita no es hoy'});
    requests.request(`citas`,'POST', cita);

}
```

Figura 4.1.2.20 Gestión de citas 3

4.1.3 Implementación de la aplicación Android

En el presente apartado se detallará el proceso de creación de la aplicación Android. Para ello, será necesario entender qué elementos conforman una aplicación Android, por qué

estados pasa durante su ejecución y qué tipos de ciclos de vida existen. Finalmente, se mostrará cómo se ha implementado cada parte de la aplicación móvil.

- **Introducción a Android.**

Una aplicación desarrollada en Android es constituida por una serie de elementos básicos de interacción con el usuario, denominados *actividades.* Además de múltiples actividades, un sistema en Android puede añadir servicios que son componentes que se ejecutan en segundo plano (background en inglés) y que realizan las mismas acciones que una actividad salvo que no poseen interfaz visible.

Un sistema Android mantiene una pila con las actividades que se han ido visualizando en el proceso, de manera que un usuario cliente de la aplicación puede volver a una actividad anterior pulsando la tecla *"volver".*

El ciclo de vida de una aplicación Android es controlado por sus actividades debido a que estas son las que van alterando la aplicación según en qué estado se sitúe cada actividad. Los posibles estados son:

- **Activo** *(Running)*. La actividad está en lo alto de la pila, por lo que es la interfaz de usuario visible.
- **Visible** *(Paused)*. La actividad es visible, pero posee otra actividad activa por encima suya. Sucede cuando una actividad no ocupa toda la pantalla o posee alguna zona transparente.
- **Parado** *(Stopped).* Cuando la actividad deja de ser visible por completo. El usuario programador es el responsable de guardar el estado de la interfaz de usuario, datos disponibles, prioridades, etc.
- **Destruido** *(Destroyed).* Sucede cuando una actividad finaliza tras invocarse el método *finish(),* o es matada (killed en inglés) por el sistema.

Cuando una actividad pasa de un estado a otro, se ciertos distintos eventos que podrán ser capturados por distintos métodos de la actividad Android. Estos pueden ser:

- **onCreate(Bundle).** Es llamado cuando una actividad es creada. En él se realizan las inicializaciones de todo tipo de datos y es el encargado de configurar la interfaz de usuario.
- **onStart().** Este método indica que la actividad está a punto de ser visible para el usuario.
- **onResume().** Este método actúa antes de entrar al estado activo, se llama cuando la actividad está a punto de interactuar con el usuario.
- **onPause().** Señala que la actividad va a pasar a segundo plano, normalmente debido a que otra actividad es lanzada a primer plano.
- **onStop().** Indica que la actividad dejará de ser visible para el usuario.
- **onRestart().** Se ejecuta para volver a representar una actividad después de pasar por el estado *onStop().*
- **onDestroy().** Este método es llamado justo antes de que la actividad en cuestión sea destruida definitivamente.

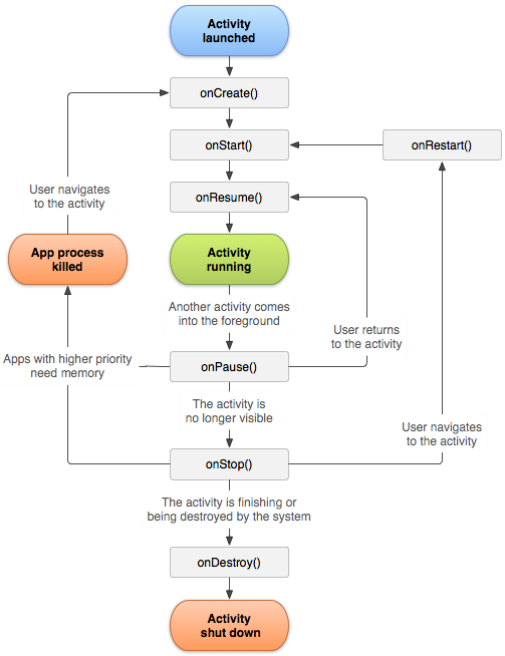


Figura 4.1.3.1 Ciclo de vida actividad Android de https://developer.android.com/guide/components/activities/activity-lifecycle?hl=es

- **Librería reutilizable.** Como se ha mencionado anteriormente, para la realización de parte de este proyecto se empleará una librería reutilizable que integra distintas clases que tramita la comunicación entre el dispositivo móvil que contiene la aplicación y los dispositivos Beacons situados en el Área Comercial. Las clases que se utilizarán de la librería serán *BeaconApplication* y *MonitoringActivity* que, a su vez, serán extendidas bajo dos clases creadas en este proyecto denominadas *BeaconApp y MonitorActivity.* Además, se han necesitado añadir funciones extras para gestionar partes de la aplicación que se desarrolla.

- *BeaconApplication.* Es la clase general de la aplicación. A partir de ella se dirigen otras actividades como *MonitoringActivity.* Esta clase es la encargada de detectar los dispositivos Beacons que, por defecto en la librería, solo busca dispositivos de tipo *altbeacon*. Además, *BeaconApplication* dispone de ciertos métodos útiles para la aplicación que se desarrolla y que serán sobreescritos y modificados en la clase *BeaconApp.*

- *MonitoringActivity.* Esta actividad es la responsable de monitorizar los distintos Beacons disponibles decidiendo si responder a dispositivos señalados bajo una región, previamente programada, o a todos los dispositivos si no existe ninguna región. Otra funcionalidad importante de esta clase es la verificación de si el dispositivo que instala la aplicación soporta la tecnología *Bluetooth Low Energy* (ver apartado 2.2). En la aplicación Android que se desarrolla, se extenderá esta clase bajo la actividad *MonitorActivity* que, por cómo está diseñada la librería, será necesario acceder a ella para verificar si el dispositivo está dentro de la región de Beacons o no lo está. Si el cliente está dentro del Área Comercial, se accederá a *MonitorActivity* para activar el método de búsqueda de beneficios y poder iniciar el proceso automático de búsqueda de privilegios personalizados.

- **BeaconApp.** Como se ha detallado en el punto anterior, esta es la actividad encargada de controlar la aplicación. En ella se han sobreescrito y modificado distintos métodos de la clase padre *BeaconApplication.*

Los métodos más importantes y útiles para este proyecto son *getEstado*, que devuelve 0 si el dispositivo está fuera de la región o 1 si el dispositivo está detectando Beacons; y *didDetermineStateForRegion* que detecta, en cada periodo de búsqueda de Beacons, en qué estado se encuentra el dispositivo. Esta última función ha sido modificada para poder realizar distintas acciones según qué actividad de la aplicación se encuentre visible en el momento en que se deja de detectar Beacons.

```
@Override
public void didDetermineStateForRegion(int state, org.altbeacon.beacon.Region arg1) {
    super.didDetermineStateForRegion(state, arg1);


    // Para saber en qué actividad está la aplicación:
    ActivityManager am = (ActivityManager) this.getSystemService(ACTIVITY_SERVICE);
    List<ActivityManager.RunningTaskInfo> taskInfo = am.getRunningTasks( maxNum: 1);
    Log.d( tag: "topActivity",  msg: "CURRENT Activity ::" + taskInfo.get(0).topActivity.getClassName());
    ComponentName componentInfo = taskInfo.get(0).topActivity; componentInfo.getPackageName();
```

Figura 4.1.3.2 Función didDetermineStateForRegion de BeaconApp 1

En la figura anterior se puede ver la primera modificación realizada al método. Se genera un manejador de Actividad para detectar, en todo momento, qué actividad se encuentra en el foco visible. Según qué actividad se encuentre activa y en qué estado se encuentre el dispositivo respecto a la detección de Beacons, se realizará una acción u otra.

```
if(state == 0){

    if(taskInfo.get(0).topActivity.getClassName().equals("com.example.appareacomercial.OfertasActivity") ||
            taskInfo.get(0).topActivity.getClassName().equals("com.example.appareacomercial.CitaCafeActivity") ||
            taskInfo.get(0).topActivity.getClassName().equals("com.example.appareacomercial.CitaSinCafeActivity") ||
            taskInfo.get(0).topActivity.getClassName().equals("com.example.appareacomercial.NoCitaActivity") ||
            taskInfo.get(0).topActivity.getClassName().equals("com.example.appareacomercial.RecomenActivity")){
        Intent intent = new Intent( packageContext: BeaconApp.this,SecondActivity.class);
        intent.putExtra( name: "email", email);
        startActivity(intent);
    }


}else if (state == 1){

    if(taskInfo.get(0).topActivity.getClassName().equals("com.example.appareacomercial.SecondActivity")){
        Intent intent = new Intent( packageContext: BeaconApp.this,SecondActivity.class);
        intent.putExtra( name: "email", email);
        startActivity(intent);
    }


}
```

Figura 4.1.3.3 Función didDetermineStateForRegion de BeaconApp 2

En la figura 4.1.3.3 se visualiza qué acciones se realizarán. Si el estado es 0, es decir, si el usuario está fuera del Área Comercial, y la actividad visible en ese instante es la que ofrece algún beneficio (ofertas, recomendaciones o gestión de citas) será necesario cambiar de actividad. Como se aprecia, la actividad a la que se irá será *SecondActivity* que es la que indica al usuario que para disfrutar de las funcionalidades de la aplicación es necesario asistir al Área Comercial. Si el estado es 1, que indica que el dispositivo detecta Beacons, y la

actividad visible es *SecondActivity* quiere decir que el usuario ha entrado al Área Comercial después de acceder a la aplicación, por lo que será necesario iniciar el proceso de búsqueda de beneficios personalizados.

- **MonitorActivity.** En el punto de la librería reutilizable ya se ha comentado la función de este método. Extendiendo a *MonitoringActivity*, si el dispositivo se encuentra dentro del Área Comercial, se invocará al método *ofertasForClient(cliente)* que iniciará el proceso automático de búsqueda de beneficios relacionados con productos disponibles en los negocios del Área Comercial.

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

    final String nombre = getIntent().getStringExtra( name: "nombre");
    final String email = getIntent().getStringExtra( name: "email");

    final List<Preference> preference = (List<Preference>) getIntent().getSerializableExtra( name: "prefe");

    BeaconApp application = ((BeaconApp) this.getApplicationContext());
    int estado = application.getEstado();
    if(clienteGuardado.equals("null")){
        if(email != null){
            clienteGuardado = email;
            if(estado == 1){

                OfertasForClient(clienteGuardado);
            }

        }
    }
```

Figura 4.1.3.4 Método onCreate de MonitorActivity

La figura 4.1.3.4 muestra el método *Oncreate* de *MonitorActivity*. En él se aprecia que, tras verificar el soporte de BLE, recoger los datos de actividades anteriores, comprobar que existe un cliente usuario y corroborar que se está detectando algún Beacon, se iniciará la función *OfertasForClient(cliente).*

En el método *OfertasForClient(cliente)*, tras realizar las distintas comprobaciones sobre el cliente enviado por parámetro y obtener los datos referentes al propio cliente, se realizará una petición HTTP con retrofit (ver apartado 2.7) para iniciar el proceso automático de Webhook enviando los datos al servidor de la aplicación. Si la respuesta del servidor es positiva (estado HTTP 200) se seguirá el proceso de forma adecuada. Por el contrario, si algo falla, se enviará un mensaje por pantalla indicando el error.

```
public void OfertasForClient(String cliente){

    String nombre = getIntent().getStringExtra( name: "nombre");
    String email = getIntent().getStringExtra( name: "email");
    List<Preference> preference = (List<Preference>) getIntent().getSerializableExtra( name: "prefe");

    if(cliente != null && cliente.equals(email)) {

        Retrofit retrofitClient = RetrofitClient.getInstance();
        iMyService = retrofitClient.create(IMyService.class);

        setDatos datos = new setDatos(nombre, email, preference);

        iMyService.createWebHookConnection(datos).enqueue(new Callback<setDatos>() {
            @Override
            public void onResponse(Call<setDatos> call, Response<setDatos> response) {

               if(response.code() == 200){
                   Toast.makeText( context: MonitorActivity.this,  text: "Buscando beneficios ...", Toast.LENGTH_SHORT).show();

               }
            }

            @Override
            public void onFailure(Call<setDatos> call, Throwable throwable) {
                Toast.makeText( context: MonitorActivity.this, throwable.getMessage(), Toast.LENGTH_LONG).show();
            }
        });

    } else{
        updateLogOfertas("Inicia Sesion para ver tus ofertas");
    }
```

Figura 4.1.3.5 Método OfertasForClient de MonitorActivity

- **MainActivity.** Siguiendo un curso normal de aplicación en ejecución, para poder utilizar las funcionalidades del sistema, será necesario desarrollar el acceso a la propia aplicación. Por ello, será necesario implementar la actividad inicial que recibirá el nombre de *MainActivity*. En esta clase, se podrá ejecutar distintos procedimientos iniciales según el caso del usuario cliente. Si el usuario dispone de un registro previo, podrá validar su acceso mediante su correo electrónico y contraseña. En el caso de no disponer de un usuario de aplicación, será indispensable realizar un registro.

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

    //Init view
    edtEmail = (EditText)findViewById(R.id.editTextTextEmailAddress);
    edtPassword = (EditText)findViewById(R.id.editTextTextPassword);

    btnRegister = (Button)findViewById(R.id.btn_register);
    btnLogin = (Button)findViewById(R.id.btn_login);

    //Init Service
    Retrofit retrofitClient = RetrofitClient.getInstance();
    iMyService = retrofitClient.create(IMyService.class);

    btnLogin.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {

            loginUser(edtEmail.getText().toString(), edtPassword.getText().toString());
        }
    });

    btnRegister.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) { handleRegisterDialog(); }
    });
```

Figura 4.1.3.6 Clase MainActivity 1

Tras inicializar los campos de texto y botones de la vista, será necesario implementar las acciones a realizar tras pulsar los distintos botones. En esta clase, se dispone de dos posibles pulsadores, el botón de registro y el de validar acceso.

Si ya se posee una cuenta de usuario, se introducirá en los cuadros de texto la información necesaria para autorizar el acceso al cliente. Como se puede apreciar en la figura 4.1.3.6, tras pulsar el botón de validar acceso (*log in* en inglés), se recogerá la información depositada por el usuario en los campos determinados y se introducirán como parámetros de entrada de la función *loginUser(email, password).*

En el caso de no disponer de un perfil vinculado a la aplicación, se pulsará el botón *Registrar Usuario* que abrirá una ventana de diálogo donde se procederá al registro.

```
private void loginUser(String email, String password) {

    if(TextUtils.isEmpty(email)){
        Toast.makeText( context: this, text: "Email cannot be null", Toast.LENGTH_SHORT).show();
        return;
    }

    if(TextUtils.isEmpty(password)){
        Toast.makeText( context: this, text: "Password cannot be null", Toast.LENGTH_SHORT).show();
        return;
    }

    iMyService.executeLogin(email, password).enqueue(new Callback<LoginResult>() {
        @Override
        public void onResponse(Call<LoginResult> call, Response<LoginResult> response) {

            if (response.code() == 200) {

                //Guardar email en Firebase RealTime Database:
                DatabaseReference ref = FirebaseDatabase.getInstance().getReference().child("email");
                ref.child("userEmail").setValue(edtEmail.getText().toString());

                Toast.makeText( context: MainActivity.this,  text: "Login Correcto", Toast.LENGTH_SHORT).show();
                goToSecondActivity();


            }else if (response.code() == 403){
                Toast.makeText( context: MainActivity.this,  text: "Contraseña incorrecta", Toast.LENGTH_SHORT).show();


            }
        }
```

Figura 4.1.3.7 Método loginUser de MainActivity

Tal y como aparece en la figura anterior, en el caso de la autorización de acceso a usuarios ya registrados, tras verificar que los campos no están vacíos, se realiza la petición HTTP por la cual se inicia el proceso de validación de acceso. Si el servidor de la aplicación indica que el proceso se ha realizado correctamente (estado HTTP 200), se mantendrá disponible la información del usuario en la base de datos de tiempo real Firebase (solo mientras el usuario esté activo en la aplicación) para poder acceder de manera rápida a los datos desde otros puntos del sistema. Además, se mostrará por pantalla un mensaje que indica que la validación es satisfactoria. En cambio, si el servidor indica que se ha producido algún fallo, se visualizará el mensaje que indica el error. En concreto, si el estado HTTP es el 403, se indicará que la contraseña introducida es incorrecta.

En el caso de iniciar un proceso de registro de usuario, se tendrá que completar la información que se requiere para crear el perfil de cliente. Estos datos son el nombre, el correo electrónico, la contraseña y qué preferencias tiene respecto a las categorías de productos disponibles en el Área Comercial.

```
private void handleRegisterDialog() {

    final AlertDialog _dialog;
    View view = getLayoutInflater().inflate(R.layout.register_layout,  root: null);
    AlertDialog.Builder builder = new AlertDialog.Builder( context: this);
    _dialog = builder.setView(view).show();

    Button btnRegister2 = (Button)view.findViewById(R.id.btn_register2);
    EditText edtNombre = (EditText)view.findViewById(R.id.idName);
    EditText edtEmail = (EditText)view.findViewById(R.id.idEmail);
    EditText edtPassword = (EditText)view.findViewById(R.id.idPassword);

    CheckBox checkBox_elect = (CheckBox)view.findViewById(R.id.checkBox_elect);
    CheckBox checkBox_hogar = (CheckBox)view.findViewById(R.id.checkBox_hogar);
    CheckBox checkBox_deportes = (CheckBox)view.findViewById(R.id.checkBox_deportes);
```

Figura 4.1.3.8 Método handleRegisterDialog del registro de usuarios

Tras rellenar los campos de registro y pulsar el botón, comprobará que los campos no estén vacíos y se implementará una forma de poder almacenar las preferencias en una lista, ya que en Android Studio no hay disponible una manera directa de crear una elección de campos (checkbox en inglés) con múltiples opciones.

```
btnRegister2.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        List<Preference> preference = new ArrayList<>();

        if(TextUtils.isEmpty(edtEmail.getText().toString())){
            Toast.makeText( context: MainActivity.this, text: "Email cannot be null", Toast.LENGTH_SHORT).show();
            return;
        }

        if(TextUtils.isEmpty(edtNombre.getText().toString())){
            Toast.makeText( context: MainActivity.this, text: "Name cannot be null", Toast.LENGTH_SHORT).show();
            return;
        }

        if(TextUtils.isEmpty(edtPassword.getText().toString())){
            Toast.makeText( context: MainActivity.this, text: "Password cannot be null", Toast.LENGTH_SHORT).show();
            return;
        }
```

Figura 4.1.3.9 Comprobación de datos en registro de usuarios

```
if(checkBox_elect.isChecked()){

    Preference pfElec = new Preference( n: "Electronica",  c: null);
    preference.add(pfElec);

}

if(checkBox_hogar.isChecked()){

    Preference pfHogar = new Preference( n: "Hogar",  c: null);
    preference.add(pfHogar);

}

if(checkBox_deportes.isChecked()){

    Preference pfDeportes = new Preference( n: "Deportes",  c: null);
    preference.add(pfDeportes);

}
```

Figura 4.1.3.10 Comprobación de selección de campos de preferencias en registro de usuarios

Posteriormente se enviará, a través de Retrofit, la solicitud HTTP de registro de usuario.

```
iMyService.executeRegister(response2).enqueue(new Callback<RegisterResult>() {
    @Override
    public void onResponse(Call<RegisterResult> call, Response<RegisterResult> response) {
        if (response.code() == 200){
            Toast.makeText( context: MainActivity.this,  text: "Registro Completado", Toast.LENGTH_LONG).show();

            _dialog.dismiss(); //Si se registra, cerramos el cuadro de registro.

            return;
        }else if(response.code() == 409){
            Toast.makeText( context: MainActivity.this,  text: "El email ya existe", Toast.LENGTH_LONG).show();
            return;
        }
    }

    @Override
    public void onFailure(Call<RegisterResult> call, Throwable t) {
        Toast.makeText( context: MainActivity.this, t.getMessage(), Toast.LENGTH_LONG).show();
    }
});
```

Figura 4.1.3.11 Petición POST de registro de usuarios

- **SecondActivity.** La función principal de esta actividad es indicar que el usuario no se encuentra en el interior del Área Comercial (no detecta Beacons), aunque dispone de otras funcionalidades como editar las preferencias del usuario registrado. Para indicar al cliente dónde se encuentra el Área Comercial, se ha incorporado un servicio de Google Maps.

```
iMyService.getUserByEmail(emailUser).enqueue(new Callback<RegisterResult>() {
    @Override
    public void onResponse(Call<RegisterResult> call, Response<RegisterResult> response) {

        if(!response.isSuccessful()){
            Toast.makeText( context: SecondActivity.this,  text: "Error " + response.code(), Toast.LENGTH_LONG).show();
            return;
        }

        user = response.body();

        nombre = user.getNombre();
        email = user.getEmail();
        preference = user.getPreference();

        String wcm = " Buenas, "+ user.getNombre();

        wellcome = findViewById(R.id.wellcome);
        wellcome.setText(wcm);


        if(estado == 1) {

            goToMonitorActivity();

        }

    }
```

Figura 4.1.3.12 Actividad SecondActivity 1

Como se ha podido comprobar en la figura 4.1.3.12, la actividad *SecondActivity*, solicita a través de una petición HTTP GET, la información del usuario que accede al sistema para poder mostrar en pantalla su nombre y, en caso de detectar Beacons, enviar los datos al

monitor de dispositivos Beacons anteriormente explicado que iniciará el proceso automático de búsqueda de beneficios personalizados.

```
// Obtain the SupportMapFragment and get notified when the map is ready to be used.
SupportMapFragment mapFragment = (SupportMapFragment) getSupportFragmentManager()
        .findFragmentById(R.id.map);
mapFragment.getMapAsync( callback: this);

@Override
public void onMapReady(@NonNull @NotNull GoogleMap googleMap) {
    mMap = googleMap;

    // Add a marker in Área Comercial and move the camera
    LatLng areaComercial = new LatLng( latitude: 36.512974553,  longitude: -4.878659040);
    mMap.addMarker(new MarkerOptions()
            .position(areaComercial)
            .title("Área Comercial JMMJ TFG"));
    mMap.setMapType(GoogleMap.MAP_TYPE_HYBRID);
    mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(areaComercial,  zoom: 15));
}
```

Figura 4.1.3.13 Integración de Google Maps en SecondActivity

En la figura anterior se aprecia cómo se ha integrado una ruta hacia el Área Comercial bajo los servicios de Google Maps. Tras crear un fragmento de mapa y sincronizarlo con la actividad de la aplicación, se han indicado las coordenadas exactas del Área Comercial para que se muestren en el mapa. Además, se le han incorporado funcionalidades de movimiento del mapa y un punto que indica el sitio al cual hay que asistir, con el título *"Área Comercial JMMJ TFG"*.

Para finalizar las funcionalidades de esta actividad, como se ha mencionado anteriormente, se podrán modificar las preferencias del perfil de usuario. Para ello, se pulsará sobre el icono de usuario de la vista y se generará una vista de diálogo similar a la del registro de usuario pero que solamente dará opción a añadir preferencias.

```
btnEditPerfil.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) { handleRegisterDialog(); }
});
```

Figura 4.1.3.14 Botón editar preferencias en SecondActivity

- **FirebaseCloudMessaging.** Es el servicio que recibe los beneficios personalizados desde el servidor de la aplicación tras el proceso de búsqueda automático. Al recibir los datos se comprueba si la aplicación está en primer plano o no. Si la aplicación está en segundo plano, se podrá acceder al beneficio mediante una notificación que será mandada al dispositivo del usuario cliente. En el caso de estar en primer plano, se enviarán los datos del beneficio concreto hacia la actividad exacta que maneje esos datos.

```java
private boolean isExecute() {
    ActivityManager activityManager = (ActivityManager) this.getSystemService(ACTIVITY_SERVICE);
    List<ActivityManager.RunningAppProcessInfo> processInfos = activityManager.getRunningAppProcesses();

    for (ActivityManager.RunningAppProcessInfo info : processInfos) {

        if (BuildConfig.APPLICATION_ID.equalsIgnoreCase(info.processName) && ActivityManager.RunningAppProcessInfo.IMPORTANCE_FOREGROUND == info.importance)
            return true;
    }
    return false;
}
```

Figura 4.1.3.15 Función que comprueba en qué plano se encuentra la aplicación

```java
JSONObject datosJSON = new JSONObject(datos);

String tipo = datosJSON.getString( name: "tipo");

if(tipo.equals("OFERTA") || tipo.equals("OFERTACOMPRA")){

    Intent intent = new Intent(getApplicationContext(), OfertasActivity.class);

    intent.putExtra( name: "datos", datos);
    Log.d( tag: "PASA1", datos);

    if(isExecute()) { // Si esta en primer plano, vamos a la siguiente actividad automaticamente.

        startActivity(intent);

    }

    PendingIntent pendiente = PendingIntent.getActivity(getApplicationContext(), requestCode: 0, intent, PendingIntent.FLAG_UPDATE_CURRENT);

    nb.setContentIntent(pendiente);

    NotificationManager nm = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

    nm.notify( id: 0, nb.build());
```

Figura 4.1.3.16 Caso de gestión de datos de ofertas en FirebaseCloudMessaging

- **OfertasActivity, RecomenActivity.** Estas actividades son las encargadas de separar los datos recibidos en formato JSON, formatearlos y mostrarlos por pantalla al usuario cliente.

Para poder visualizar imágenes a través de la recepción de una URL, será necesario utilizar los servicios de *Picasso* que se habrá tenido que registrar previamente en el fichero *build.gradle* del proyecto Android Studio.

```java
String datos = getIntent().getStringExtra( name: "datos");
JSONObject datosJSON = new JSONObject(datos);
String tipo = datosJSON.getString( name: "tipo");
String nomOf = datosJSON.getString( name: "producto");
String url = datosJSON.getString( name: "url");
String pA = datosJSON.getString( name: "precioAntes");
String pD = datosJSON.getString( name: "precioActual");
String tienda = datosJSON.getString( name: "negocio");

tipoOferta = findViewById(R.id.tipoOferta);
nombreT = findViewById(R.id.tiendaNombre);
nombreT.setText("Solo disponible en " + tienda);

if(tipo.equals("OFERTA")){
    tipoOferta.setText("Has obtenido una oferta según tus preferencias");
}else if (tipo.equals("OFERTACOMPRA")){
    tipoOferta.setText("Has obtenido una oferta según tus compras");
}

nombreOferta = findViewById(R.id.nombreOferta);
nombreOferta.setText(nomOf);
```

```
precioAntes = findViewById(R.id.precioAntes);
precioAntes.setPaintFlags(precioAntes.getPaintFlags() | Paint.STRIKE_THRU_TEXT_FLAG);
precioAntes.setText(pA+"€");

precioDespues = findViewById(R.id.precioActual);
precioDespues.setText(pD+"€");
```

Figura 4.1.3.17 Recepción, formateo y muestra de los datos disponibles en OfertasActivity

```
producto = findViewById(R.id.productoImagen);

Picasso.with(this) Picasso
        .load(url) RequestCreator
        .resize( targetWidth: 1000,  targetHeight: 1000)
        .centerCrop()
        .into(producto);
```

Figura 4.1.3.18 Carga de imagen a través de URL en OfertasActivity

Otra funcionalidad que dispone estas actividades es ocultar el beneficio mostrado, para ello, se podrá seleccionar un botón que cierra la actividad. Antes de cambiar de actividad, se mostrará un cuadro de alerta para confirmar que se rechaza el beneficio o que se cancela la acción.

```
cancelButton = findViewById(R.id.cancelBoton);

cancelButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        AlertDialog.Builder builder = new AlertDialog.Builder( context: OfertasActivity.this);
        builder.setTitle("¡Alerta!");
        builder.setMessage("¿Seguro que quieres rechazar la oferta?")
                .setPositiveButton( text: "SÍ", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {

                        //Obtenemos el email necesario de Firebase para volver a actividades anteriores:
                        db = FirebaseDatabase.getInstance().getReference();
                        db.child("email").addValueEventListener(new ValueEventListener() {
                            @Override
                            public void onDataChange(@NonNull DataSnapshot snapshot) {

                                if (snapshot.exists()) {

                                    String email = snapshot.child("userEmail").getValue().toString();
                                    Intent intent = new Intent(getApplicationContext(), CentralActivity.class);
                                    intent.putExtra( name: "email", email);

                                    startActivity(intent);
                                    dialog.dismiss();
                                }
                            }

.setNegativeButton(android.R.string.cancel, new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        dialog.dismiss();
    }
}).show();
```

Figura 4.1.3.19 Rechazar beneficio con mensaje de alerta en OfertasActivity

Otra característica que destacar es la implementación de la lista de productos en la actividad *RecomenActivity*. Los datos que se reciben en formato *arrayJSON* se convertirá al

formato lista a través de la librería *Gson* que se ha instalado en el proyecto a través del fichero *build.gradle*. Posteriormente a la conversión, se creará un *listView* donde a través de un adaptador, se representarán todos los productos en una lista que mantiene el mismo formato.

```
JSONArray productosArray = datosJSON.getJSONArray( name: "productos");
String prodArrayString = productosArray.toString();

Gson gson = new Gson();
Type produtType = new TypeToken<List<Producto>>(){}.getType();
List<Producto> productos = gson.fromJson(prodArrayString,produtType);

listView = findViewById(R.id.lvLista);

listView.setAdapter(new RecomenAdapter( context: this, productos));
```

Figura 4.1.3.20 Conversión de datos y creación de la lista de productos listView en RecomenActivity

```
public class RecomenAdapter extends BaseAdapter {

    private  static LayoutInflater inflater = null;
    Context contexto;
    List<Producto> productList;

    public RecomenAdapter(Context context, List<Producto> productList) {
        this.contexto = context;
        this.productList = productList;
        inflater = (LayoutInflater) context.getSystemService(context.LAYOUT_INFLATER_SERVICE);
    }

    @Override
    public int getCount() { return productList.size(); }

    @Override
    public Object getItem(int position) { return null; }

    @Override
    public long getItemId(int position) { return 0; }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {

        final View vista = inflater.inflate(R.layout.elemento_lista,  root: null);

        TextView producto =  vista.findViewById(R.id.prodList);
        TextView categoria = vista.findViewById(R.id.categoriaProd);
        TextView precio = vista.findViewById(R.id.precioProd);
        String precioString = String.valueOf(productList.get(position).getPrecio());
        ImageView imagen = vista.findViewById(R.id.imagenProducto);
        String url = productList.get(position).getUrl();
```

```
producto.setText(productList.get(position).getProducto());
categoria.setText(productList.get(position).getCategoria());
precio.setText(precioString + " €");


Picasso.with(contexto) Picasso
        .load(url) RequestCreator
        .resize( targetWidth: 400,  targetHeight: 400)
        .centerCrop()
        .into(imagen);



return vista;
}
```

Figura 4.1.3.21 Adaptador para la lista de productos recomendados

En la figura 4.1.3.21 se visualiza como se ha implementado el adaptador para la lista de productos recomendados, donde a través de una vista de diseño *InflaterLayout* se ha generado una estructura que seguirá cada elemento de la *listView* de productos que se mostrará.

- **CentralActivity.** Tras rechazar el beneficio ofrecido por el proceso automático realizado al detectar algún Beacon, la aplicación dirige su foco visible a esta Actividad. En *CentralActivity* se dispone de 3 botones que representan las acciones fundamentales que se realizan. La primera acción posible será, como en el caso de *SecondActivity*, la modificación de las preferencias seleccionadas por el usuario cliente. La segunda funcionalidad posible es la de regresar al beneficio mostrado anteriormente. Finalmente, la última acción posible será indicar al sistema que el usuario ha llegado a su cita, por lo que se realizará una gestión de las citas disponibles para el cliente y se le mostrará la información personalizada adecuada a su situación personal.

```
btnBeneficios.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        //Se obtienen los datos para volver al beneficio:
        db = FirebaseDatabase.getInstance().getReference();

        db.child("profit").addValueEventListener(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                if (snapshot.exists()) {

                    String datos = snapshot.child("userProfit").getValue().toString();

                    try {

                        JSONObject datosJSON = new JSONObject(datos);

                        String tipo = datosJSON.getString( name: "tipo");

                        //Log.d("TIPO", tipo);

                        if(tipo.equals("OFERTA") || tipo.equals("OFERTACOMPRA")){

                            Intent intent = new Intent(getApplicationContext(), OfertasActivity.class);
                            intent.putExtra( name: "datos", datos);
                            startActivity(intent);
```

Figura 4.1.3.22 botón para volver al beneficio personalizado en CentralActivity

En la figura 4.1.3.22 se aprecia cómo se recupera el beneficio personalizado que se ha ofrecido al cliente con anterioridad. Para ello, se accede a los datos del beneficio, previamente guardados en RealDatabase de Firebase (solo durante la sesión activa del usuario). Según qué tipo de beneficio sea, se regresará a una actividad u otra.

Si se selecciona el botón de cita, se indicará al sistema que se ha llegado en ese instante a la cita y que se está esperando turno. Para ello, se creará una variable con la fecha y hora exactas al pulsar el botón y se enviará junto al correo electrónico del usuario al servidor de la aplicación para iniciar un proceso de Webhook que gestionará la cita del cliente.

```
btnCitas.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        String fechaLlegada = new SimpleDateFormat( pattern: "yyyy/MM/dd HH:mm:ss").format(Calendar.getInstance().getTime());

        CitasCall citasCall = new CitasCall(emailUser, fechaLlegada);

        iMyService.setLlegadaCita(citasCall).enqueue(new Callback<CitasCall>() {
            @Override
            public void onResponse(Call<CitasCall> call, Response<CitasCall> response) {

                if(response.code() == 200){
                    Toast.makeText( context: CentralActivity.this,  text: "Buscando cita ...", Toast.LENGTH_SHORT).show();

                }

            }

            @Override
            public void onFailure(Call<CitasCall> call, Throwable t) {
                Toast.makeText( context: CentralActivity.this, t.getMessage(), Toast.LENGTH_LONG).show();
            }
        });
    }
});
```

Figura 4.1.3.23 botón cita en CentralActivity

- **CitaCafeActivity, CitaSinCafeActivity, NoCitaActivity.** Estas son las actividades que indican las distintas posibilidades que posee un usuario al seleccionar que ha llegado a su cita. Su función principal es la de mostrar al cliente la información respectiva a su cita (ver apartado 4.1.2 sección de citas). En el caso de llegar a tiempo a su cita, el cliente será recompensado con un café que podrá solicitar en la cafetería del Área Comercial a través de un código QR. En el caso disponer de cita registrada pero no se ha asistido en el día exacto, se informará la fecha de cita.

En estas actividades, al igual que en *OfertasActivity* y *RecomenActivity* se dispone de un botón para volver a *CentralActivity*. Antes de poder regresar a la actividad central, se mostrará un panel de alerta que será necesario confirmar para poder salir de la actividad actual.

```
btnVolver = findViewById(R.id.volver);

btnVolver.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        AlertDialog.Builder builder = new AlertDialog.Builder( context: CitaCafeActivity.this);
        builder.setTitle("¿Seguro que quiere volver?");
        builder.setMessage("Podría perder su café si no lo ha registrado")
                .setPositiveButton( text: "SÍ", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {

                        //Obtenemos el email necesario de Firebase para volver a actividades anteriores:
                        db = FirebaseDatabase.getInstance().getReference();
                        db.child("email").addValueEventListener(new ValueEventListener() {
                            @Override
                            public void onDataChange(@NonNull DataSnapshot snapshot) {

                                if (snapshot.exists()) {

                                    String email = snapshot.child("userEmail").getValue().toString();
                                    Intent intent = new Intent(getApplicationContext(), CentralActivity.class);
                                    intent.putExtra( name: "email", email);

                                    startActivity(intent);
                                    dialog.dismiss();
                                }
                            }
```

Figura 4.1.3.24 botón volver en actividades que muestran información de citas

4.2 Pruebas

Tras la etapa de diseño e implementación del sistema, se realizará una fase de ensayos que pongan a prueba la arquitectura creada con el fin de detectar posibles errores y verificar el correcto funcionamiento.

Se separarán las pruebas en los dos sectores que engloban la totalidad del proyecto. En primer lugar, se realizarán pruebas en la parte de los servidores implementados y, por otra parte, se pondrá a prueba la aplicación Android (parte cliente).

4.2.1 Pruebas en servidores

Para la realización de pruebas a nivel de servidor, se utilizará la plataforma *Postman* (ver apartado 2.11). Esta plataforma se utilizará como cliente que enviará peticiones a los servidores que se han implementado y así comprobar, tanto por la salida generada en *Postman* como por la consola integrada en el servidor (con *Visual Studio Code*), que las distintas funcionalidades creadas en los servidores funcionan de manera correcta.

La primera prueba que se realizará será la validación de acceso de un posible usuario en la aplicación. Si realizamos la solicitud con datos de un correo electrónico erróneo la salida generada es la siguiente:

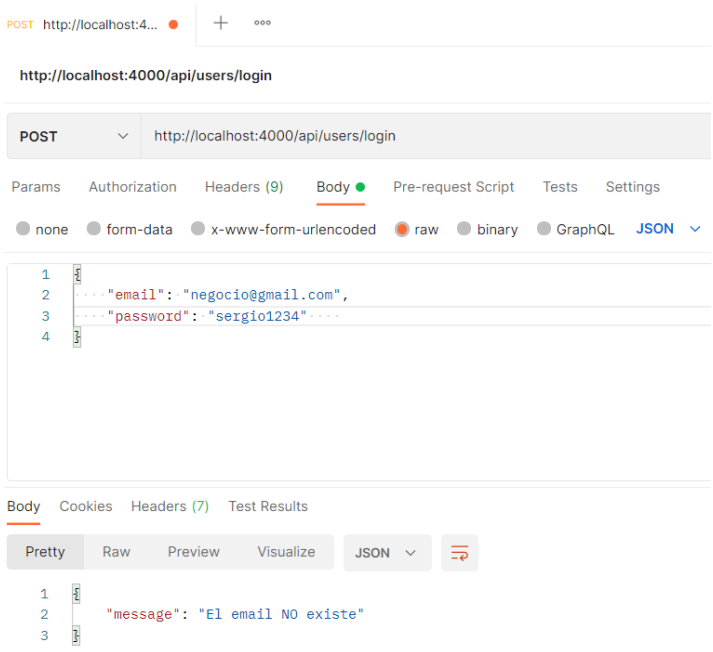


Figura 4.2.1.1 Prueba validar acceso de usuario 1

En la figura anterior se aprecia como se genera un mensaje que indica que ese correo electrónico no existe.

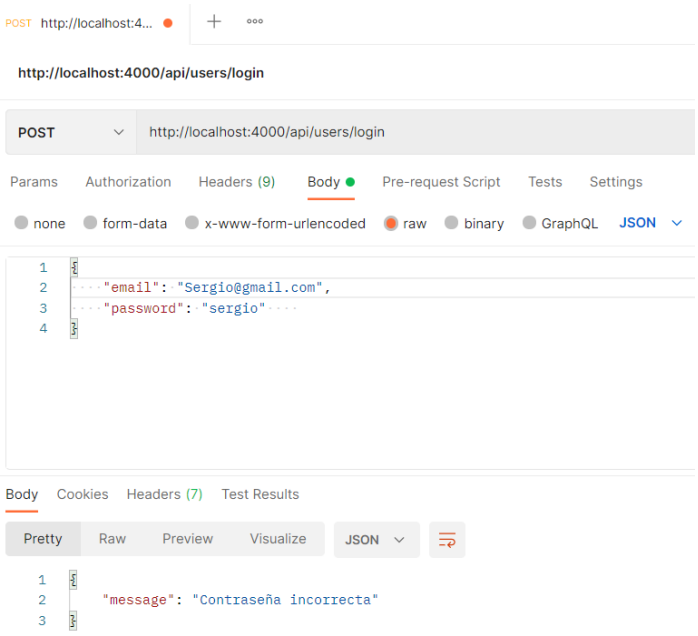


Figura 4.2.1.2 Prueba validar acceso de usuario 2

La figura 4.2.1.2 muestra que, en el caso de insertar una contraseña errónea, el servidor muestra un mensaje de alerta de contraseña incorrecta.

Finalmente, si se inserta un correo electrónico válido y su contraseña vinculada, se autorizará el acceso correctamente.

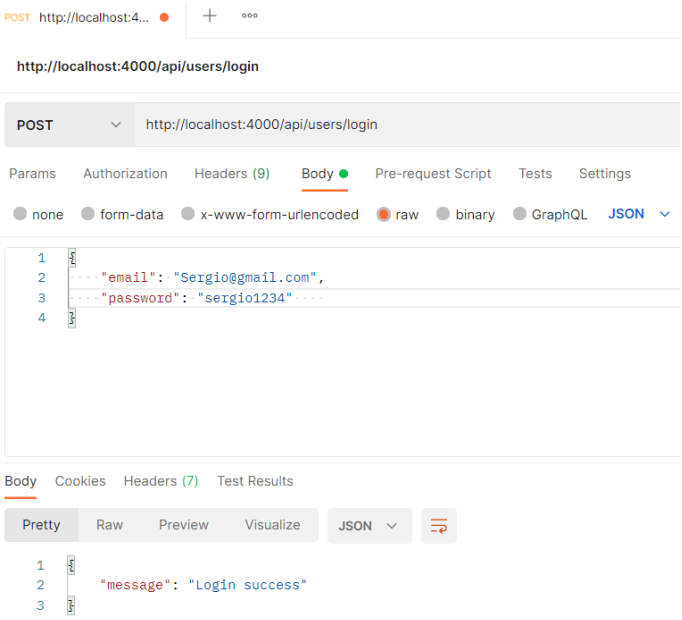


Figura 4.2.1.3 Prueba validar acceso de usuario 3

Otra prueba fundamental para el sistema es el registro de usuarios. Si insertamos en *Postman* un cuerpo (body en inglés) válido de registro, se puede apreciar que se realiza de manera correcta el registro de usuarios.

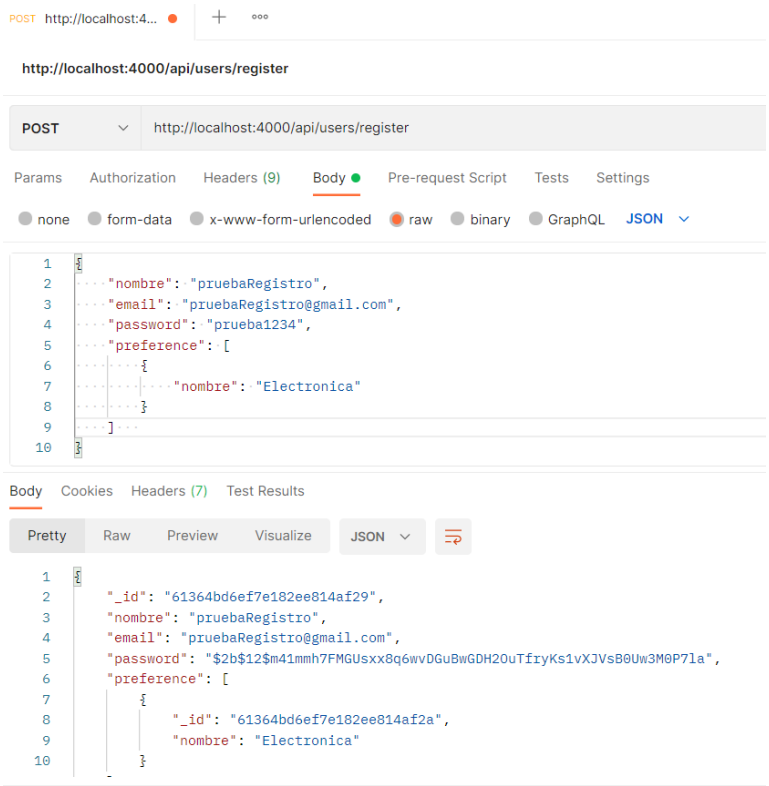


Figura 4.2.1.4 Prueba registro de usuarios 1

Para comprobar que el usuario se ha creado de manera satisfactoria, se puede acceder a la información disponible en la base de datos mongo DB a partir de la aplicación *MongoDB*

Compass, que representa de manera gráfica qué datos se disponen en la base de datos en cuestión.

En la siguiente figura, se visualiza el usuario anteriormente registrado:

Figura 4.2.1.5 Prueba registro de usuarios 2

Además, como se ha explicado anteriormente, en la aplicación desarrollada es posible modificar las preferencias. Para probar esto, se modificarán las preferencias del usuario creado en la figura 4.2.1.4.

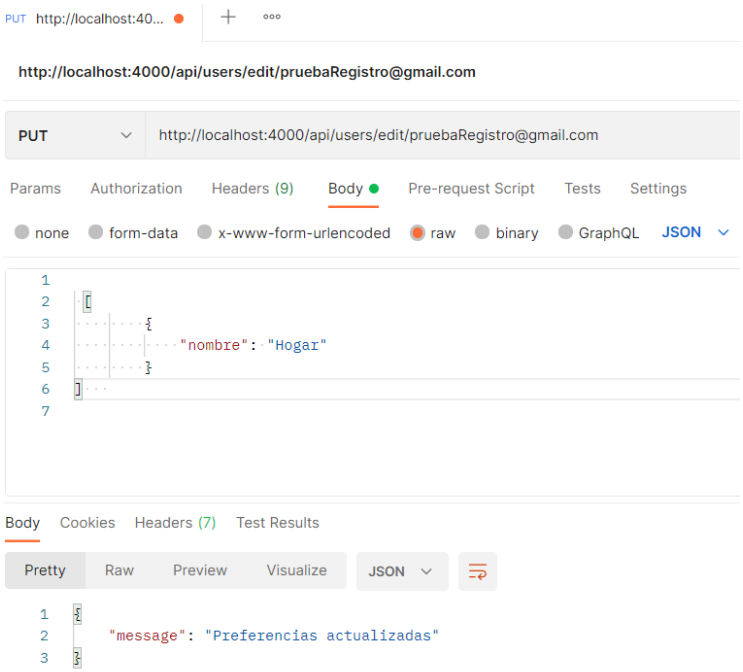


Figura 4.2.1.6 Prueba Modificar Preferencias 1

En la figura anterior se observa que tras el envío de la solicitud PUT con las preferencias modificadas, se recibe un mensaje que indica que las preferencias se han actualizado correctamente. Si se observa la información del usuario en *MongoDB Compass* se podrá corroborar que se han actualizado correctamente.

```
_id: ObjectId("61364bd6ef7e182ee814af29")
nombre: "pruebaRegistro"
email: "pruebaRegistro@gmail.com"
password: "$2b$12$m41mmh7FMGUsxx8q6wvDGuBwGDH2OuTfryKs1vXJVsB0Uw3M0P7la"
preference: Array
  0: Object
      _id: ObjectId("6136501bef7e182ee814af35")
      nombre: "Hogar"
__v: 0
```


Figura 4.2.1.7 Prueba Modificar Preferencias 2

Para finalizar las pruebas de servidores, se realizará el ensayo de la funcionalidad fundamental del sistema. Se probará el proceso automático de búsqueda de beneficios que interconecta al servidor de la aplicación con el servidor de negocio implementado. A través de una solicitud POST con el envío de la información del cliente se iniciará el proceso automático.

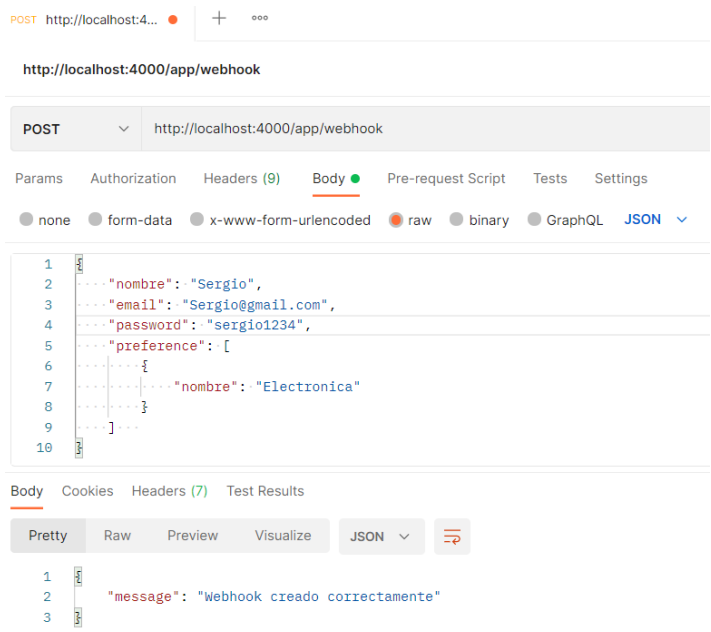


Figura 4.2.1.8 Prueba Proceso Webhook 1

En la anterior figura se aprecia que el proceso se ha iniciado correctamente. Para ver el resultado del procedimiento realizado, será necesario inspeccionar las consolas de los servidores ya que por ella se imprimirán las acciones que se han realizado en cada uno de los puntos. Tras iniciar el Webhook, en el servidor del negocio se realizará la gestión de la información y se generará un beneficio personalizado. En la siguiente figura se visualizará el resultado en este caso de prueba:

Figura 4.2.1.9 Prueba Proceso Webhook 2

En la figura 4.2.1.9 se comprueba que la información ha sido recibida correctamente, se ha gestionado y se ha elaborado un beneficio tipo *"OFERTA"*. Este beneficio, siguiendo el procedimiento automático, se enviará de vuelta al servidor de la aplicación. Si se visualiza el terminal de este servidor, se apreciará lo siguiente:

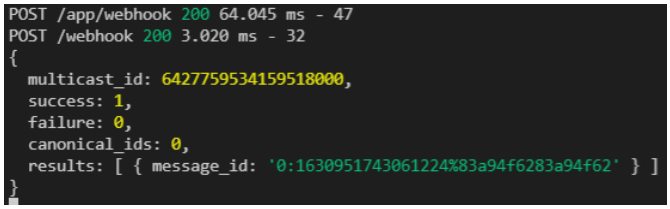


Figura 4.2.1.10 Prueba Proceso Webhook 3

La imagen anterior muestra que tanto la solicitud inicial como la recepción del beneficio se han realizado de manera satisfactoria. Además, se aprecia que, a través de *Firebase Push Notification* se han enviado correctamente los datos a la aplicación.

4.2.2 Pruebas en aplicación cliente

Desde el punto de vista de la aplicación Android, luego de su implementación, se ha realizado un proceso de pruebas cuyo objetivo es encontrar defectos y evaluar la calidad del producto final. En esta fase, se han utilizado varios métodos como caja blanca (White-box testing en inglés), caja negra (Black-box testing en inglés) y pruebas individuales con usuarios finales.

Se denomina caja blanca a las pruebas realizadas a partir del conocimiento del código y la estructura del producto. Estas pruebas son aplicadas a través de la cognición de la arquitectura desarrollada y las tecnologías empleadas para ello. Las pruebas que se llevarán a cabo bajo este método son referentes a la relación entre la lógica de negocio desarrollada y las restricciones que el sistema impone para poder establecer un formato común entre las distintas partes del sistema. Tras un amplio análisis en esta fase de pruebas, se ha detectado que la relación entre la lógica de negocio y la aplicación Android establecen una comunicación satisfactoria.

Las pruebas de caja negra son las realizadas mediante el análisis de la información que entra al sistema y las respuestas finales que se producen, sin tener en cuenta el funcionamiento interno. Estas son las pruebas que se han realizado mediante el uso de las funcionalidades básicas desde la propia aplicación. Por ejemplo, si se registra un usuario con sus datos de entrada, la salida esperada es el registro en la base de datos mongo DB como usuario afiliado al sistema.

Para finalizar esta fase, se han realizado otras pruebas mediante la creación de usuarios individuales. Estos ensayos son generados desde el punto de vista de la persona que ha diseñado la aplicación, por lo que se intenta, a partir de las pruebas, corroborar que acciones concretas de funcionalidades especificas son resueltas de manera correcta.

En la siguiente tabla se representarán las pruebas de mayor importancia que han sido realizadas:

Prueba	Salida	Tipo	Funcionamiento
Registro	Registro de usuario en mongo DB	Caja negra	Correcto
Registro sin completar algún campo	No permite el registro	Usuario individual	Correcto
Registro sin formato email	No permite el registro	Usuario individual	Correcto
Registro de usuario ya existente	No permite el registro	Usuario individual	Correcto
Validar acceso	Permite el inicio de sesión en el sistema	Caja negra	Correcto
Validar acceso sin completar algún campo	No permite el inicio de sesión	Usuario individual	Correcto
Validar acceso con algún dato erróneo	No permite el inicio de sesión	Usuario individual	Correcto
Editar preferencias	Modifica los datos	Caja negra	Correcto
Obtener beneficio concreto	Se visualiza el beneficio en la app	Caja blanca	Correcto
Obtener tipo de cita concreta	Se visualizan los datos de la cita	Caja blanca	Correcto
Volver al beneficio	Se regresa al mismo beneficio ofrecido	Usuario individual	Correcto

Figura 4.2.2.1 Tabla de pruebas de aplicación

5

Conclusiones y trabajo futuro

5.1 Conclusiones

A modo de conclusión, se puede decir que se ha cumplido con el objetivo principal que se planteó inicialmente en este Trabajo de Fin de Grado, desarrollando una solución integral para el desarrollo de aplicaciones de comercio electrónico inteligente basada en el uso de dispositivos Beacons que cumple con todos los requisitos definidos. Para ello, se ha implementado un conjunto significativo de las funcionalidades más comunes en este tipo de negocios.

Además, se han resuelto de manera satisfactoria todos los objetivos que se determinaron al principio de este proyecto. El servidor de aplicación desarrollado ofrece las funcionalidades esenciales de un servidor de comercio inteligente. En este servidor reside la lógica que aporta a la aplicación Android las acciones necesarias para ofrecer un comercio personalizado e inteligente de cara al usuario cliente final, iniciando procesos automáticos de búsqueda de beneficios y estableciendo conexiones con servidores de otros negocios que pueden integrarse al sistema bajo un registro en este mismo servidor de comercio inteligente. Otro objetivo era crear un componente de negocio concreto que aporte una lógica implementada para el comercio y el servidor necesario para el propio negocio. Como prueba de la resolución satisfactoria de este objetivo se puede comprobar que la aplicación y su servidor establecen una comunicación efectiva y satisfactoria en la obtención de todo tipo de beneficios personalizados (como ofertas, recomendaciones y citas). Por lo que, no solo se ha implementado correctamente el componente señalado como segundo objetivo, sino que se ha integrado de manera adecuada a las restricciones impuestas en el servidor de comercio inteligente. Como último objetivo suplementario, se ha ampliado la infraestructura común existente modificando ciertos métodos que se definen en la librería utilizada para adecuar las funcionalidades de la infraestructura a una arquitectura de una dimensión considerable como es la que se ha desarrollado en este trabajo.

Atendiendo al desarrollo técnico del proyecto, es necesario destacar que la mayoría de las tecnologías aplicadas en el trabajo son desconocidas para el alumnado de este grado, ya que algunas de ellas son muy novedosas (caso de tecnología Webhook, Firebase o Beacon), otras no se han proporcionado bajo el contenido académico de las distintas asignaturas (caso de desarrollo en Android, Retrofit o servidores) y algunas se han expuesto con un contenido altamente teórico o, en su defecto, poco contenido práctico (caso de BBDD no SQL o API REST). Por ello, fue difícil sintetizar todas las ideas o requisitos que se habían propuesto inicialmente. Una vez resueltas las primeras fases de planificación, análisis y diseño, las etapas posteriores fueron encauzadas de manera rápida, destacando la importancia de las etapas anteriormente mencionadas.

Finalmente, hemos de destacar las ventajas que este Trabajo de Fin de Grado aporta. Es evidente que tanto las tecnologías aplicadas como los dispositivos de IoT son muy demandados en la actualidad debido a su constante evolución y su aporte en la sociedad actual. Por ello, y teniendo en cuenta nuestro futuro laboral próximo, el presente proyecto será un hito importante para nuestros inicios como ingeniero informático, pues como dice Bill Gates, *"La clave del éxito en los negocios está en detectar hacia dónde va el mundo y llegar ahí primero" (William Henry Gates III, 2014).*

5.2 Trabajo futuro

Debido a la constante evolución de las tecnologías aplicadas, la multitud de funcionalidades que ofrece un comercio inteligente y la cantidad de recursos que se pueden añadir a una aplicación Android, este proyecto resulta muy fácilmente extensible. Además, debido a que para la realización de este sistema se utiliza una infraestructura común (librería) ofrecida por otro trabajo anterior, si esa librería es mejorada, modificada o extendida, a su vez, el trabajo futuro de este sistema puede ser aumentado también. Algunas ideas que se pueden aportar para un trabajo futuro serían las siguientes:

- **Integrar la obtención de otros beneficios.** La tecnología Webhook junto a las funcionalidades que aporta los dispositivos Beacons hacen que la aplicación pueda integrar otro tipo de recursos dentro del mismo ámbito de comercio inteligente. Ejemplo de esto podría ser la utilización de los Beacons existentes en el Área Comercial para crear una ruta a tiempo real (tipo GPS, pero por tecnología BLE) para guiar al usuario cliente hacia una zona determinada del recinto (negocios, áreas de descanso, reuniones, etc.).

- **Seguridad entre la aplicación y su servidor.** Como se ha podido comprobar en los distintos apartados de esta memoria, en el sistema se han añadido distintos tipos de seguridad en determinadas zonas de la arquitectura. Se ha proporcionado seguridad entre el servidor de la aplicación y los servidores de los negocios a partir del uso de claves API. Además, se ofrece seguridad en el servidor de comercio inteligente respecto al almacenamiento de contraseñas, las cuales son procesadas por encriptación y *"hashing"*. Para un sistema más seguro, podría aportarse seguridad entre la aplicación y su servidor, por ejemplo, con el uso de OAuth 2.0.

- **Mejoras en la librería (infraestructura común).** Debido a la gran extensión de este Trabajo de Fin de Grado ha sido imposible poder resolver o mejorar ciertos aspectos que son derivados del proyecto que se realizó con anterioridad a este que se está llevando a cabo. Un aspecto por mejorar sería el cambio de *MonitoringActivity* a *MonitoringService*, es decir, la actividad podría ser contemplada bajo un servicio que realizaría las mismas acciones que la misma, pero sin necesidad de tener una interfaz gráfica de usuario. En las pruebas que se realizaron en el Trabajo de Fin de Grado anterior, se utilizaba la interfaz gráfica de este monitor para poder visualizar en qué momento se detecta un Beacon y cuándo deja de ser visible. En una aplicación de entorno real, como es la que se ha implementado, visualizar esta actividad *(MonitorActivity)* en el proceso de pasar desde la interfaz de inicio a la actividad que muestra el beneficio obtenido, es algo que debería de obviarse para optimizar la visualización de la aplicación.

- **Migrar servidores a la red.** En este Trabajo de Fin de Grado, se ha implementado una arquitectura completa de comercio inteligente aplicable a entornos reales. En esta etapa de desarrollo del sistema, los servidores implementados son alojados en el propio equipo (localhost). Si se desease publicar la aplicación o ponerla en fase real de producción, sería necesario migrar los servidores a algún *hosting* o a la nube (servicio *Cloud*).

Referencias

[1]. Introducción a Android Studio | Desarrolladores de Android, https://developer.android.com/studio/intro?hl=es-419

[2]. ¿Qué es la Tecnología Beacon? – Ryte Digital Marketing Wiki, https://es.ryte.com/wiki/Beacon

[3]. What Are Beacons and How Beacons Technology Works? – Intellectsoft, https://www.intellectsoft.net/blog/what-are-beacons-and-how-do-they-work/

[4]. Node.js – Introduction – Tutorialspoint, https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm

[5]. Firebase: qué es, para qué sirve, funcionalidades y ventajas, https://www.digital55.com/desarrollo-tecnologia/que-es-firebase-funcionalidades-ventajas-conclusiones/

[6]. Cliente-Servidor – Wikipedia, La enciclopedia libre, https://es.wikipedia.org/wiki/Cliente-servidor#:~:text=La%20red%20cliente%2Dservidor%20es,vez%20que%20estos%20son%20solicitados.

[7]. Transferencia de Estados Representacional – Wikipedia, La enciclopedia libre, https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional

[8]. Máster en Desarrollo de Aplicaciones Android – Ciclo de vida de actividad, http://www.androidcurso.com/index.php/tutoriales-android/37-unidad-6-multimedia-y-ciclo-de-vida/158-ciclo-de-vida-de-una-actividad

[9]. Introducción a las actividades Android – Desarrolladores de Android, https://developer.android.com/guide/components/activities/intro-activities?hl=es

[10]. View – Android Developers, https://developer.android.com/reference/android/view/View#attr_android:contentDescription

[11]. Date – JavaScript en español, https://lenguajejs.com/javascript/fechas/date-fechas-nativas/

[12]. Retrofit, https://square.github.io/retrofit/

[13]. Claves de api – Documentación de IBM, https://www.ibm.com/docs/es/mfci/7.6.1?topic=components-api-keys

[14]. ¿Cómo securizar tus APIS con Oauth? – Paradigma, https://www.paradigmadigital.com/dev/oauth-2-0-equilibrio-y-usabilidad-en-la-securizacion-de-apis/

[15]. Caja Blanca vs Caja Negra – Tester Moderno, https://www.testermoderno.com/caja-blanca-vs-caja-negra/

Apéndice A

Manual de instalación

Este apartado servirá como guía para poder utilizar el sistema implementado o realizar trabajos futuros sobre él. Inicialmente, será necesario disponer de unos requisitos previos que serán de vital importancia para instalar la arquitectura creada. Estas condiciones son:

- Disponer de un dispositivo Android cuya versión de sistema operativo sea igual o superior a 5.0 (Lollipop).
- Obtener un equipo local para poder realizar las instalaciones del *back-end* del sistema.
- Instalar un editor de código como Visual Studio Code para visualizar la codificación de los servidores y poder ejecutarlos desde la terminal.
- Instalar Android Studio para importar el proyecto de aplicación Android.
- Instalar en el equipo local mongo DB.
- Obtener los ficheros de los servidores locales y la aplicación Android.

Tras comprobar que se cumplen todos los requisitos mencionados anteriormente, se procederá a la importación de las distintas partes del proyecto.

En cuanto a los servidores, estos poseen en su interior ficheros de configuraciones esenciales para que funcionen de manera correcta, ya que disponen de todos los módulos Node.js instalados en el propio proyecto. Por ello, solamente será necesario importar los servidores en dos proyectos separados:

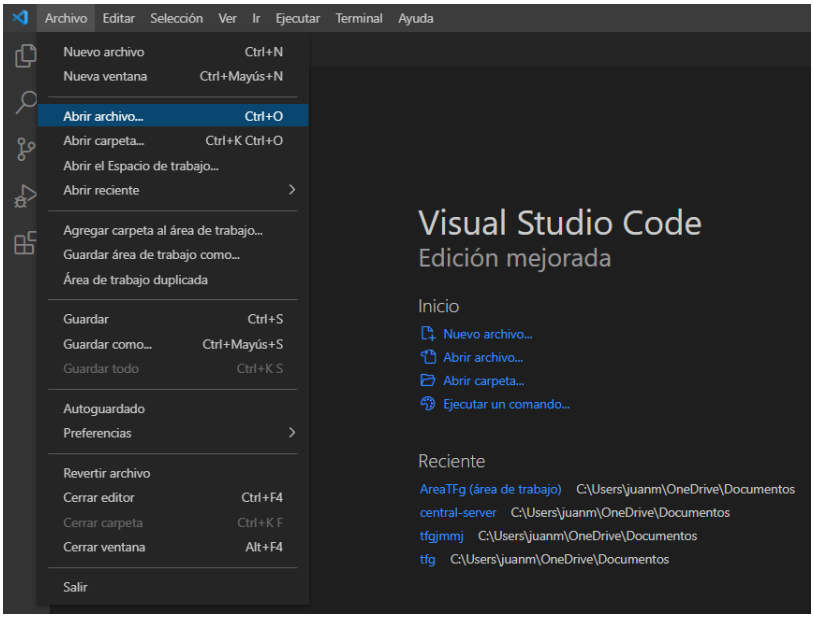


Figura 1, Apéndice A. Importar proyectos

Una vez importados los servidores será necesario iniciarlos, para ello se le ha facilitado el trabajo al usuario que desee instalar el sistema a partir de un módulo de Node.js denominado *Nodemon***.** A partir de él y la generación de un script, para poder ejecutar los distintos servidores solo será necesario escribir en la terminal (dentro de cada ruta específica), *"npm run dev".*

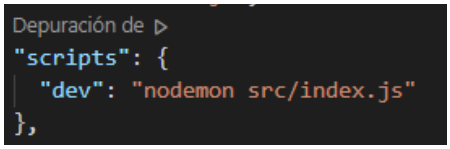


Figura 2, Apéndice A. Script procedente de *package.json* de los distintos servidores

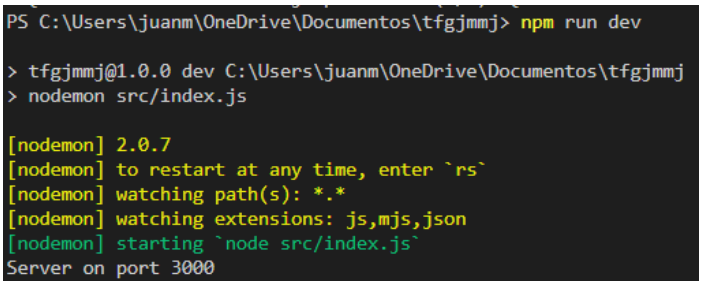


Figura 3, Apéndice A. Inicio de servidor

Para que la base de datos mongo DB funcione de manera correcta será necesario iniciar el servicio desde la ruta donde se aloja la instalación de la propia base de datos. Para dispositivos *Windows*, la ruta por defecto es *"C:\Program Files\MongoDB\Server\4.4\bin".* Si el usuario se encuentra en el interior de esta carpeta, la siguiente acción a realizar será iniciar el ejecutable denominado "*mongod".*

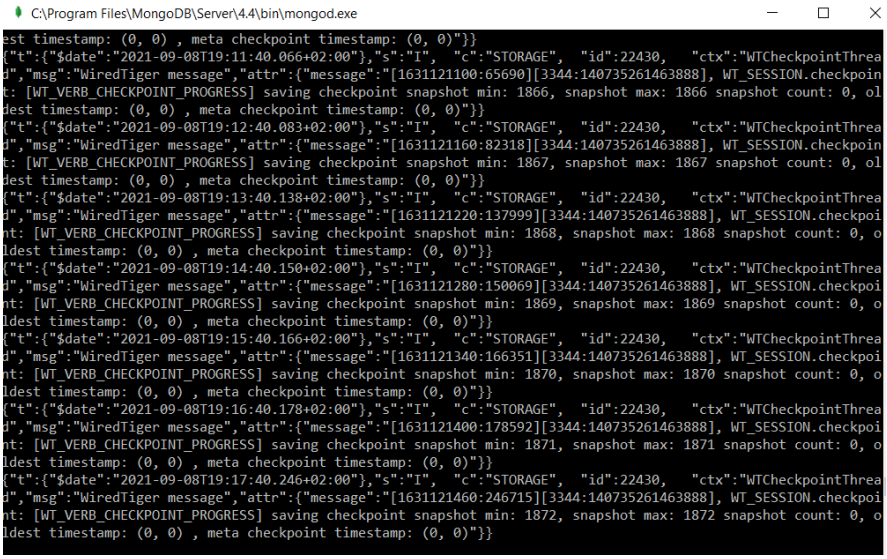


Figura 4, Apéndice A. Ejecución de mongod

Posteriormente al inicio de los servidores y la base de datos No SQL, será necesario importar el proyecto de la aplicación a Android Studio. Este proyecto ya contiene todas las

funcionalidades detalladas en la memoria y la importación de la infraestructura común reutilizada para las conexiones entre los Beacons y el dispositivo móvil. Para poder abrir el proyecto, dentro de Android Studio, se seleccionará "*File"* y luego "*Open"*. Luego, será necesario indicar al sistema dónde se encuentra el fichero que contiene el proyecto en cuestión. Seleccionamos el proyecto, indicamos *"Ok"* y se importará el proyecto de manera satisfactoria.

Llegados a este punto, solo faltaría instalar la aplicación en el dispositivo Android. Para ello, primero se ha de realizar en dicho dispositivo algunas configuraciones previas que harán que el propio instrumento sea visible para Android Studio. La primera acción que realizar será activar en el dispositivo las opciones de desarrollador. Para ello, será necesario dirigirse hacia el panel de ajustes dentro de su dispositivo Android y posteriormente a la opción *"Acerca del teléfono".* Una vez de acceda, habrá que ir hasta la opción *"Información de software"* donde se pulsará reiteradas veces en *"Número de compilación"* para activar el modo de desarrollador.

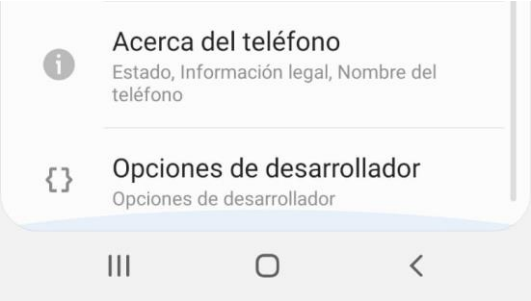


Figura 5, Apéndice A. Opciones de desarrollador en dispositivo Android

Dentro de las configuraciones del modo de desarrollador, es necesario activar la opción denominada *"Depuración USB"* para que a través de la conexión USB entre el equipo local y el dispositivo móvil sea Android Studio capaz de detectar a este último instrumento.

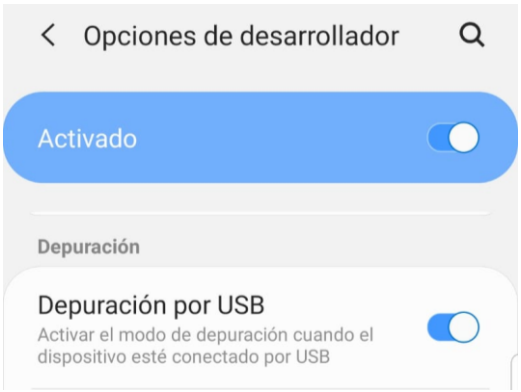


Figura 6, Apéndice A. Depuración USB en dispositivo Android

Por último, si se accede a Android Studio se comprobará que ya es visible el dispositivo móvil para poder ejecutar la aplicación. Por ello se añadirá la dirección IP pública para poder establecer conexión con el servidor de comercio electrónico (en fichero *RetrofitClient*) y seleccionando el botón *Run* (botón verde de inicio), se instalará la aplicación de manera satisfactoria en su dispositivo.

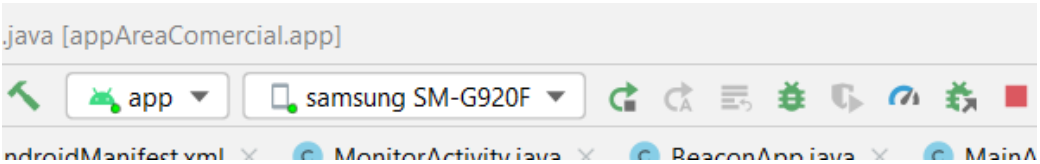


Figura 7, Apéndice A. Instalación de aplicación Android en dispositivo móvil

Apéndice B

Manual de usuario

En este apéndice se mostrará una sencilla guía con todas las acciones y funcionalidades posibles que un usuario cliente de la aplicación Android podría llevar a cabo.

En primer lugar, la vista inicial da dos posibles acciones según la situación del cliente. Si es un usuario nuevo, para poder disfrutar de las funcionalidades del sistema será necesario pulsar en la opción *Registrarse* que dirigirá al usuario hacia el panel de registro. Por el contrario, si ya se posee un usuario en la aplicación, será necesario autorizar el acceso mediante la inserción del correo electrónico y contraseña vinculados al perfil de registro.

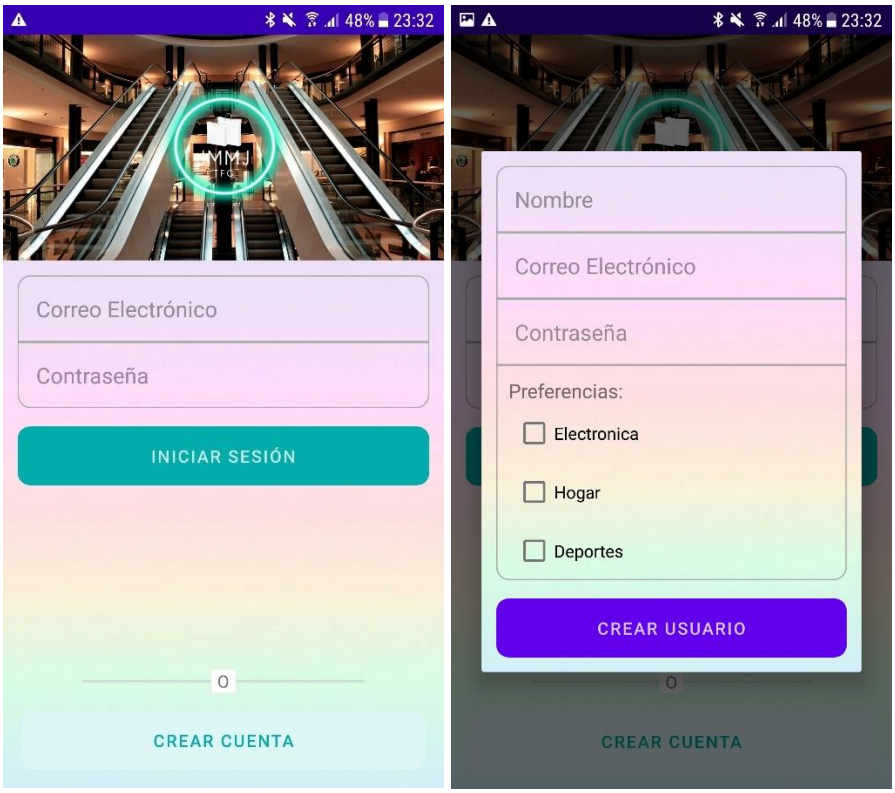


Figura 1, Apéndice B. Vista inicial y registro de la aplicación Android

Una vez se ha accedido a la plataforma, según donde se encuentre el usuario situado, se direccionará hacia una interfaz u otra. Si el cliente está en el interior del Área Comercial detectando algún dispositivo Beacon, la aplicación, de manera automática, mostrará al usuario un beneficio personalizado (oferta o recomendación de productos).

Figura 2, Apéndice B. Beneficio tipo OFERTA aplicación Android

Si se desea salir de este apartado, el usuario tendrá que pulsar en el botón de salida (botón de cruz situado en el centro inferior de la pantalla). Será necesario confirmar que se quiere abandonar esta vista.

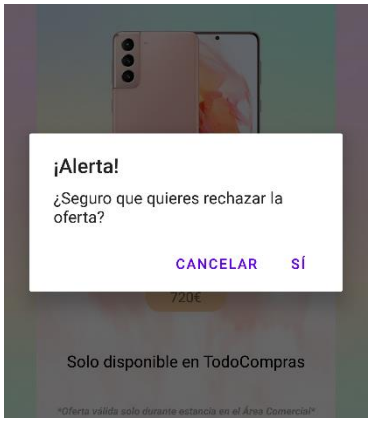


Figura 3, Apéndice B. Alerta para rechazar beneficio aplicación Android

Llegados a este punto, el usuario visualizará la actividad central de la aplicación. A partir de ella se podrá volver al beneficio ofrecido anteriormente, gestionar posibles citas o modificar las preferencias que el usuario desee.

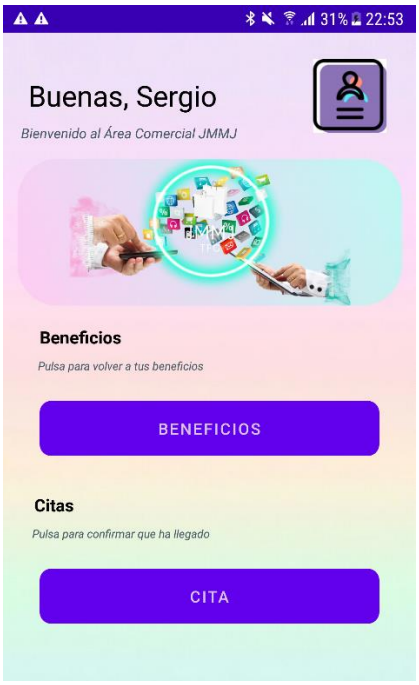


Figura 4, Apéndice B. Vista central de aplicación Android

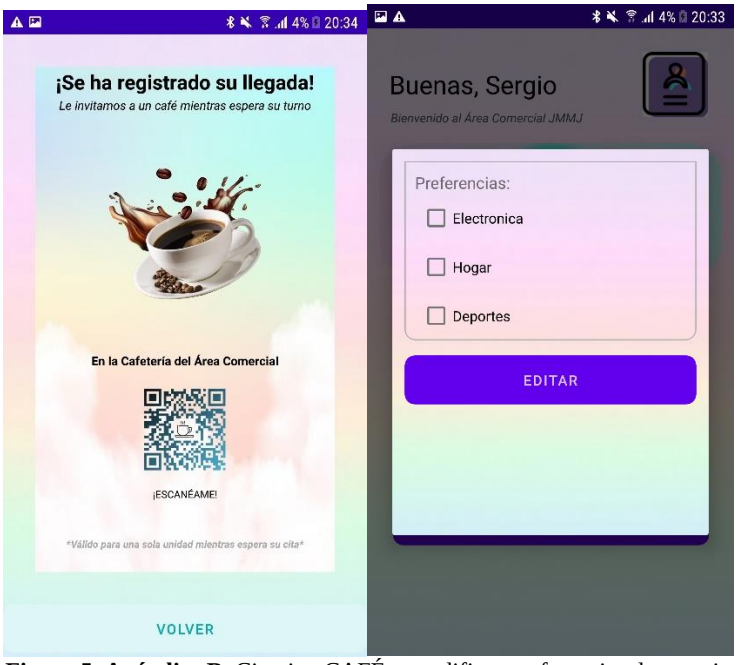


Figura 5, Apéndice B. Cita tipo CAFÉ y modificar preferencias de usuario

Por otra parte, si el usuario no se encuentra en las instalaciones del Área Comercial, apreciará en la aplicación un mensaje de alerta y un mapa para poder dirigir al cliente hacia el Área Comercial, en el caso de que desee disfrutar de las funcionalidades que aporta la aplicación instalada. Desde esta vista, también será posible modificar las preferencias del usuario registrado.

Figura 6, Apéndice B. Vista cuando no detecta ningún Beacon aplicación Android