Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Creación de una plataforma para la interacción geolocalizada con el mundo físico

Trabajo Fin de Máster

Máster Universitario en Ingeniería Informática

Autor: Morro Ibáñez, Álvaro

Tutores: Fons Cos, Joan

Pelechano Ferragud, Vicente

Curso 2020/2021

Resumen

En el presente trabajo de fin de máster se ha desarrollado un prototipo de aplicación orientada a la actividad turística que, usando los conceptos de la inteligencia ambiental y el Internet de las cosas, permite interactuar con recursos del mundo físico al proporcionar información sobre objetos o áreas geolocalizadas dependiendo de la posición del usuario. La solución desarrollada proporciona de forma continua a los usuarios información sobre los recursos geolocalizados que estén en sus cercanías de forma clara, concisa y accesible. La solución propuesta en este documento permite que organizaciones, empresas u otras entidades puedan ofrecer a sus usuarios información geolocalizada de manera sencilla y con un bajo coste creando para estos una experiencia más satisfactoria

Palabras clave: Inteligencia Ambiental, IoT, Geolocalización, Android, museos, realidad aumentada.

Resum

En el present treball de fi de màster s'ha desenvolupat un prototip d'aplicació orientada a l'activitat turística que, usant els conceptes de la intel·ligència ambiental i la Internet de les coses, permet interactuar amb recursos del món físic en proporcionar informació sobre objectes o àrees geolocalitzades depenent de la posició de l'usuari. La solució desenvolupada proporciona de manera contínua als usuaris informació sobre els recursos geolocalitzats que estiguen en la seua rodalia de manera clara, concisa i accessible. La solució proposada en aquest document permet que organitzacions, empreses o altres entitats puguen oferir a les seues usuaris informació geolocalitzada de manera senzilla i amb un baix cost creant per a aquests una experiència més satisfactòria

Palabras clave: Intel·ligència Ambiental, IoT, Geolocalització, Android, museus, realitat augmentada.

Abstract

In this master's thesis, a prototype of an application oriented to tourist activity has been developed, which, using the concepts of environmental intelligence and the Internet of Things, allows the interaction with resources of the physical world providing information on objects or geolocated areas depending on the position of the user. The developed solution continuously provides users with information about geolocated resources that are in their vicinity in a clear, concise and accessible way. The solution proposed in this document allows organizations, companies or other entities to offer their users geolocated information in a simple way and at a low cost, creating for them a more satisfactory experience.

Keywords : Ambient Intelillence, IoT, Geolocalization, Android, museums, augmented reality.

Tabla de contenidos

1. Introducción

Desde la llegada de los llamados teléfonos inteligentes, la transformación digital no ha hecho más que acelerarse con nuevas formas de comunicarnos y relacionarnos con otras personas y con el mundo que nos rodea. Hoy en día tenemos en la palma de nuestra mano acceso a información de manera prácticamente infinita, servicios que se nos proporcionan desde cualquier lugar del mundo y de forma instantánea a cualquier hora del día, algo impensable hace tan solo unas décadas.

Junto con el avance en la disponibilidad y la cantidad de los contenidos, se han producido también cambios en la forma en la que nos relacionamos con los dispositivos que permiten este acceso rápido e inmediato a ingentes cantidades de información. Los primeros dispositivos móviles aparecidos a finales del siglo veinte eran poco más que enormes baterías a las que se acoplaban los dispositivos que permitían la comunicación inalámbrica. Uno de estos primeros dispositivos era el Motorola Dynatac 8000x que pesaba unos ochocientos gramos y medía más de treinta centímetros de alto.

Ilustración 1: Teléfono Motorola Dynatac 8000X

En apenas un par de décadas los dispositivos habían reducido su tamaño de forma considerable a la vez que incrementaban sus prestaciones y ofrecían nuevos servicios a sus usuarios. Sin embargo la forma de relacionarnos con estos dispositivos seguía siendo a través de teclados físicos e interfaces gráficas. Habría que esperar hasta finales de la década de los dos mil, para ver la llegada de una nueva forma de interacción hombre-máquina: la pantalla táctil.

Este cambio de paradigma, de botones físicos a controles virtuales, permitió que se pudiera interactuar con los dispositivos de una manera mucho más intuitiva y natural.

En los años siguientes, esta progresión hacia dispositivos en los que cada vez es menos necesario interactuar físicamente con ellos no ha hecho más que acrecentarse, no es raro encontrar hoy en día aparatos con control por voz, o que disponen de algún tipo de control mediante gestos. Estas nuevas formas de interactuar con la tecnología que nos rodea hace que comunicarse con los dispositivo electrónicos y hacer uso de sus servicios es cada vez más fácil y cómodo haciendo que la tecnología pase cada vez más desapercibida y se funda con nuestro entorno.

En este contexto es donde surge la idea de la Inteligencia Ambiental, conseguir que los computadores se encuentren en todas partes, inadvertidas y siempre disponibles, haciendo que podamos desarrollar nuestras actividades accediendo a información gracias a estos dispositivos que, idealmente, serán capaces de interpretar las diferentes situaciones y contextos para dar una respuesta acorde.

1.2 Motivación

Como hemos visto la Inteligencia Ambiental es una de las tecnologías llamadas a revolucionar la forma en la que usamos y nos relacionamos con la tecnología. Es por ello que este trabajo surge del interés por explorar posibles aplicaciones de la inteligencia ambiental en el mundo real, más allá de aplicaciones de carácter científico o experimental.

Además se ha decidido enfocar el trabajo sobre la actividad turística que es una de las actividades económicas más importantes de nuestro país amén de ser una actividad con la que, en menor o mayor medida, todos estamos familiarizados.

La sector turístico, como cualquier otro sector económico, está sujeto a continuos avances tecnológicos e innovaciones que buscan mantener una oferta turística atractiva ofreciendo nuevos servicios que mejoren las experiencias de sus usuarios.

Por ejemplo, en el año dos mil dieciocho los Museos Vaticanos comenzaron un proceso de digitalización junto con la compañía española Minsait para ofrecer a sus visitantes una aplicación que les permita conocer en tiempo real la afluencia en cada una de las salas de la institución permitiéndoles evitar esperas, acceder a información detallada sobre las obras que se exponen en museo o ver promociones adaptadas a sus gustos.

Este tipo de iniciativas permite mejorar la experiencia de los visitantes ofreciendo algo distinto e innovador lo que puede suponer un incremento en el número de visitas y un aumento en la satisfacción de los visitantes.

Pero los visitantes y usuarios de estas atracciones turísticas no son los únicos que se benefician de las nuevas tecnologías. Las propias atracciones obtiene múltiples beneficios mediante el empleo de estas. Por ejemplo en el caso de los museos, “uno de sus mayores desafíos es comprender el patrón de interacción entre los visitantes y las exhibiciones o el propio mueso “ (Pierdicca, Roberto et al ,2019; 4) para lo cual resulta indispensable conocer cómo se desplaza la gente dentro del museo, que salas son las más visitadas o en cuales los visitantes están más tiempo.

Por lo tanto, resulta evidente que la aplicación de tecnologías como el Internet de las Cosas o la Inteligencia Ambiental suponen un gran salto cualitativo tanto para instituciones turísticas y culturales como para los propios usuarios de las mismas.

Ilustración 2: Varias capturas de la aplicación del museo BMW donde se muestra información sobre el museo y las colecciones.

1.3 Objetivos

Este proyecto final de máster persigue los siguientes objetivos:

- Explorar las las posibles aplicaciones de la Inteligencia ambiental en un caso práctico centrado sobre la actividad turística.
- Desarrollar un prototipo de una plataforma que permita a los usuarios interactuar con el mundo físico accediendo información sobe el lugar que están visitando. Esta solución constará de un servidor web con los datos de las diferentes localizaciones y una aplicación móvil que permita acceder a dicha información.
- Explorar la idoneidad del uso del protocolo *MQTT* para aplicaciones de Inteligencia Ambiental

1.4 Alcance del Proyecto

El presente proyecto pretende crear una solución tecnológica para estudiar la posible aplicación de la inteligencia ambiental a ámbitos prácticos relacionados con la interacción con el mundo físico, en concreto se pretende desarrollar un prototipo de plataforma que permita a los usuarios obtener información sobre recursos geolocalizados en tiempo real dependiendo de su posición.

La solución planteada consta de un servidor web desarrollado en JAVA y una aplicación móvil Android, prestando especial atención en que la interacción del usuario con el sistema sea la mínima imprescindible. Además, a la hora de desarrollar este sistema se han tenido en cuenta diversos aspectos de la inteligencia ambiental para comprobar la viabilidad de los mismos en un caso real.

Por último, comentar que el trabajo desarrollado no puede considerarse una aplicación completa ya que como veremos en este documento, no se han desarrollado todos los aspectos de la misma aunque puede servir de base para futuros desarrollos

1.5 Metodología y Plan de desarrollo

Para el desarrollo del presente trabajo se ha seguido una metodología basada en *sprints* o fases de manera que se ha dividido las tareas a realizar en varias fases con las mínimas dependencias entre ellas. De esta manera se ha podido llevar a cabo el desarrollo de la solución propuesta de una manera más ágil.

Se han identificado cinco fases a llevar a cabo en este proyecto: Estudio previo del estado del arte, tecnologías a emplear y caso de estudio seleccionado, diseño de la solución, desarrollo del servidor, desarrollo de la aplicación móvil, creación de la memoria del trabajo.

Estas fases a su vez están compuestas por una serie de tareas que se enumeran a continuación:

- Estudio previo:
  - Estado de las tecnologías actuales
  - Caso de estudio
  - Tecnologías a emplear
- Diseño de la solución:
  - Diseño de la arquitectura general
  - Diseño del modelo de datos
  - Diseño de la arquitectura del servidor
  - Diseño de la arquitectura de la aplicación móvil
  - Diseño de la base de datos
  - Diseño de la comunicación entre los distintos componentes
- Desarrollo del servidor:
  - Base de datos
  - Modelo de datos
  - Gestor de comunicaciones
  - Servicios

        - Servicio de persistencia
        - Servicio JSON
        - Servicio MQTT
        - Socket Factory
        - Servicio de propiedades
- Desarrollo de la aplicación:
    - Interfaz de usuario:
        - Main activity
        - Detalle Activity
        - Ajustes Activity
        - Mapa Activity
    - Modelo de datos
    - Servicios:
        - Servicio de imágenes
        - Servicio JSON
        - Servicio MQTT
        - Servicio en segundo plano
        - Gestor de mensajes
- Creación de la memoria

En cuanto a la planificación de las fases arriba descritas, esta se ha hecho teniendo en cuenta las dependencias que hay entre algunas de las tareas. De esta manera, en primer lugar se ha llevado a cabo la fase de estudio previo la cual es indispensable para poder acometer el resto de fases. Una vez finalizada esta primera fase se ha seguido con el desarrollo del servidor y de la aplicación móvil. Aunque para poder emplear la aplicación es necesario que el servidor esté desarrollado, hay partes del desarrollo de la misma que no lo necesitan como puede ser la creación de la interfaz de usuario.

Por ello, las fases de desarrollo del servidor y de la aplicación se han llevado a cabo de manera simultánea. Por último, en paralelo a todas estas fases se ha trabajado en la memoria de forma continua completándola según avanzaba el desarrollo de la solución.

En el siguiente diagrama se muestra la planificación de las diferentes fases junto con sus tareas asociadas:

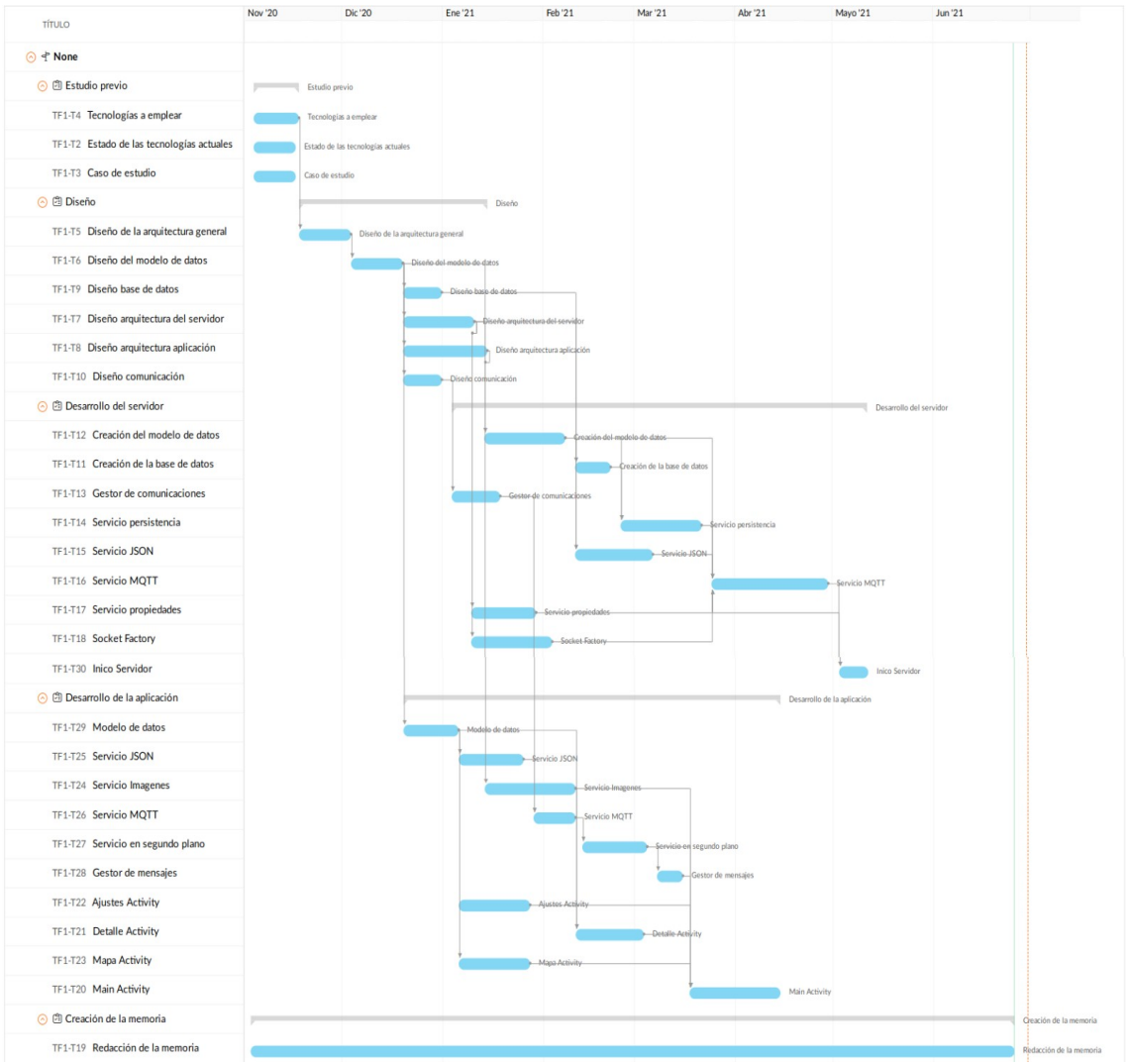

Ilustración 3: diagrama de Gantt mostrando la planificación de las tareas del proyecto

1.6 Estructura de la Memoria

El presente documento está estructurado de la siguiente manera:

En primer lugar, después de una breve introducción explicando el contexto en el cual se desarrolla este proyecto, comenzaremos describiendo que es la Inteligencia Ambiental, veremos ejemplos de la misma así como de otros conceptos relacionados con esta tecnología.

Una vez cubierto este tema, pasaremos a describir las principales tecnologías implicadas en la solución desarrollada así como alternativas que se han considerado a las mismas. También se describirán los paradigmas y patrones de programación utilizados a la hora de desarrollar la solución.

A continuación se analizará el estado del arte y el caso de estudio propuesto para después describir la solución que se propone. Veremos su diseño y componentes, haciendo hincapié en como ha afectado al mismo las ideas de la Inteligencia Ambiental. Se explicará su funcionamiento así como el proceso de instalación y puesta en marcha además de la infraestructura necesaria para su funcionamiento.

Comentaremos por separado la implementación del servidor y de la aplicación móvil explicando su diseño y funcionalidad así como las interacciones que se producen entre ellas. Se prestará especial atención a los principales componentes tanto del servidor como de la aplicación explicando en detalle la implementación de los mismos

Por último, veremos una guía en la que se explicará como instalar y utilizar el software desarrollado, las conclusiones del proyecto y los posibles desarrollos futuros que podrían llevarse a cabo tomando como base el presente trabajo.

2. Contexto tecnológico

En los últimos años venimos asistiendo a cambios de gran calado en el mundo de la tecnología, la irrupción del internet de las cosas, más conocido por sus siglas en inglés (*IoT*), ha hecho que los dispositivos conectados a internet sean ahora mucho más comunes que hace una década. Estos dispositivos, pueden ser desde un reloj o una bombilla hasta una máquina industrial, nos permiten recopilar una cantidad inmensa de datos prácticamente en cualquier lugar pero, incluso más importante, podemos interactuar con ellos gracia a su conexión a la red.

De esta manera, controlar las luces o las persianas del hogar desde el teléfono móvil ha pasado de ser un hecho prácticamente de ciencia ficción a una realidad cada vez más habitual. En este contexto, se han venido desarrollando una serie de tecnologías y paradigmas que han hecho posible esta revolución tecnológica.

En este capitulo, vamos a describir algunas de estas tecnologías las cuales se han empleado en el desarrollo de este proyecto, pero antes, veamos algunos ejemplos de uso de estas tecnologías en el ámbito de la actividad turística.

Una de las posibles aplicaciones de este tipo de tecnologías consiste en ofrecer a los visitantes de una exposición o un evento la posibilidad de obtener información adicional directamente en su teléfono móvil sobre el contenido de la exhibición. Típicamente esto se consigue gracias a un sistema de sensores capaces de detectar la posición del usuario y transmitir esta información a uno a varios servidores los cuales mandarán la información deseada al *smartphone* del visitante.

Este es el caso del castillo de Maschio Angioino situado en Nápoles, Italia. Durante una exposición en el castillo, se implementó un sistema basado en sensores *Blueethot* que ofrecía información sobre las obras expuestas a los visitantes al acercarse a las mismas. En este caso se utilizaros sensores basados en la placa *BeagleBone Black* con un coste por unidad de unos cuarenta dólares.

Estos sensores utilizan la tecnología inalámbrica *Bluethoot* para buscar dispositivos móviles en sus cercanías y, una vez encontrados, enviarles información sobre una obra de arte en concreto.

Según se explica en la conclusiones del estudio de este sistema, “La satisfacción general manifestada por los voluntarios hacía la infraestructura fue positiva con una puntuación media de 7.43 [...] En cuanto a la utilidad, los usuarios coincidieron en que la aplicación fue útil en general [...], facilitando hasta

cierto punto la adquisición de un mejor conocimiento [...] y una visión más profunda [...] sobre la obra de arte expuesta" (Piccialli et All, 2014, p. 5).

Como vemos este tipo de proyectos puede ayudar a incrementar la satisfacción de los usuarios de una atracción turística. Sin embargo existen algunas limitaciones que hacen que estos proyectos no sean válidos para una gran cantidad de situaciones. Por ejemplo, el uso de sensores o balizar *Bluethoot* encarecen en gran medida la implantación de este tipo de sistemas, además, en entornos a la intemperie, estos dispositivos pueden no ser los adecuados debido a que necesitan una infraestructura para funcionar (redes eléctricas y de telecomunicaciones) que pueden no estar disponibles.

Debido a estos inconvenientes que pueden presentar el uso de sensores y, buscando desarrollar una solución de bajo coste y lo más accesible posible, se han decido no emplear dispositivos de geolocalización además de utilizar tecnologías probadas y gratuitas para evitar riesgos y rebajar todo lo posible el coste de la solución.

Pero además de tecnologías en las últimas décadas se viene desarrollando y estudiando toda una serie de ideas, teorías y paradigmas que pretender dar forma a la revolución que supone vivir rodeados de dispositivos "inteligentes". Uno de los paradigmas más importantes es el de la **Inteligencia Ambiental.**

2.1 La Inteligencia Ambiental

Antes de entrar a hablar en profundidad sobre la Inteligencia Ambiental, detengámonos por un momento en el concepto de "inteligencia". La Real Academia Española de la Lengua, define la inteligencia como la "capacidad de entender o comprender" (Real Academia Española, s.f., definición 1) y como la "capacidad de resolver problemas" (Real Academia Española, s.f., definición 2).

Si atendemos a estas definiciones, observamos que para considerar a algo o alguien inteligente, este debe ser capaz de entender y comprender los problemas que se le plantean así como de resolverlos; por lo que el mero tratamiento de la información de forma automatizada no pude considerarse como una muestra de inteligencia. Por lo tanto, ¿podemos realmente llamar inteligentes a los dispositivos que usamos en nuestro día a día como son los teléfonos móviles, *tablets* o computadores?

La respuesta es no, el adjetivo "inteligente" aplicado a este tipo de dispositivos se debe más a razones de mercadotecnia que a describir adecuadamente las características de los mismos.

A principios de la década de los dos mil, comenzaron a aparecer en el mercado dispositivos de telefonía con capacidad de conectarse a Internet y que disponían de múltiples funciones como agenda, navegador, reloj o cámara fotográfica. Con cada nueva generación, se incluían más y mejores característica y servicios como por ejemplo, los asistentes virtuales. Todo esto ha contribuido a crear la ilusión de que nos encontramos ante verdaderos dispositivos inteligentes.

Desgraciadamente, esto no es cierto y en la gran mayoría de los casos, estos dispositivos simplemente realizan la tarea para la que han sido programados pero no son capaces de conocer y comprender el contexto en la que esta se desarrolla.

Por ejemplo: si registramos un nuevo evento en la agenda de nuestro teléfono, digamos que para una cita médica, nuestro dispositivo nos avisará de este evento igual que de cualquier otro, pero poco más.

Si este fuera realmente inteligente, nos indicaría a que hora tenemos que salir de casa teniendo en cuenta circunstancias del entorno como el tráfico o el sitio al que vamos, también podría, avisarnos si nos surge un compromiso ese mismo día e incluso podría cambiarnos la cita a otro momento.

Según Cook, Augusto y Jakkula (2007), la idea básica detrás de la inteligencia ambiental, es que enriqueciendo un entorno con tecnología (sensores y dispositivos interconectados por una red), se puede construir un sistema que actúe como un “mayordomo electrónico”, el cual percibe las características del usuario y su entorno para luego razona sobre los datos acumulados y, finalmente, selecciona que acciones llevar a cabo que beneficien al usuario en ese entorno.

Como podemos ver con el ejemplo anterior la inteligencia ambiental va mucho más allá del simple tratamiento de los datos sino que requiere que los sistemas sean conscientes del contexto y sean capaces de adaptarse y reaccionar de acuerdo a los cambios que se producen en este.

Por lo tanto, podemos definir la inteligencia ambiental como un paradigma tecnológico en el cual los dispositivos de computación están integrados en el entorno, siempre disponibles y que son capaces de adaptarse e incluso anticiparse a las necesidades de los usuarios.

Podemos observar que la Inteligencia Ambiental está relacionada con muchas otras tecnologías como por ejemplo:

- Dispositivos Embebidos: Los dispositivos de computación deben estar integrados en el entorno que nos rodea, pasando desapercibidos.

- Consciencia del Contexto: Los dispositivos deben reconocer el contexto en el que se encuentran para poder ofrecer soluciones adecuadas.
- Anticipación y predicción: Los dispositivos no solo deben actuar de forma reactiva sino que deben hacerlo de forma anticipada, actuando antes de que el usuario lo pida para conseguir adaptarse a los cambios en el entorno.
- Interacción natural: Permitir que los usuario puedan interactuar con los dispositivos de una forma natural a través de gestos o de la voz, haciendo de la interacción un hecho más fácil e intuitivo.

2.1.1 Dispositivos Embebidos

Ilustración 4: Nevera Family Hub.
Fuente:Samsung

Como se ha comentado anteriormente, un sistema construido alrededor de la idea de la Inteligencia Ambiental, debe desaparecer de la vista de los usuarios y pasar a formar parte del entorno. La miniaturización de los computadores ha hecho que hoy en día los tengamos prácticamente en cualquier sitio, en nuestro coche, televisión o nevera los dispositivos de computación han ido reduciendo su tamaño haciendo que estos se puedan colocar en cualquier sitio sin que su presencia sea notoria a simple vista.

Gracias a esto, cualquier objeto puede convertirse en un dispositivo conectado capaz de ofrecernos servicios que hace unos años eran impensables, como por ejemplo, poder encargar la compra directamente desde nuestra nevera.

2.1.2 Consciencia del Contexto y Anticipación

Un dispositivo de computación consciente de su contexto (donde se encuentra y que está ocurriendo a su alrededor) es aquel que “se adapta dependiendo de su localización, de la gente a su alrededor, del resto de dispositivos accesibles así como a los cambios que se producen en estas cosas a lo largo del tiempo” (Schilit et All, 1994, p. 1). Esta capacidad de comprender y adaptarse a los cambios es lo que hace que estos dispositivos sean realmente inteligentes y que ayuden de una forma más natural y precisa a sus usuarios con sus tareas y necesidades.

Si un dispositivo es capaz de conocer que está está realizando el usuario y cuales son las características del entorno donde se desarrolla esa actividad, podrá darle información relevante y precisa que le resulte útil en ese momento

Por ejemplo mostrando rutas alternativas cuando el usuario está conduciendo por una carretera con in tráfico muy denso, encendiendo y apagando las luces de una casa en función de los movimientos de sus habitantes o encendiendo la calefacción si se detecta una bajada de las temperaturas en una casa domótica.

Una aproximación para conseguir que un sistema de Inteligencia Ambiental sea capaz e anticiparse a los cambios en su entorno consiste en que el sistema aprenda poco a poco a partir de los datos que recaba de su entorno. Siguiendo con el ejemplo de la casa domótica, esta podría aprender sobre las rutinas de sus inquilinos con el paso del tiempo, detectando que zonas de la casa usan con más frecuencias o a que horas se utilizan más ciertas habitaciones, etcétera.

2.1.3 Interacción Natural

Otro aspecto fundamental de la Inteligencia Ambiental es la forma en la que los usuarios se relacionan con los diversos dispositivos que componen el sistema. Dado que los sistemas deben ser capaces de actuar según lo que percibe de su contexto, “como un mayordomo observa que las actividades se desarrollan con la expectativa de ayudar cuando (y solo si) se necesita” (Diane J. Cook et al, 2007, p.12), la interacción con los mismos debe producirse de la manera más natural posible.

Para conseguir esto, se debe evitar emplear interfaces físicas como pantallas y teclados sustituyéndolos por técnicas de interacción como el reconocimiento de voz o gestos, es decir deberíamos interactuar con ellos del modo más natural posible.

Que la interacción se realice sin tener que usar interfaz física (pantalla, teclado, botonera,...) permite una interacción más cómoda y al alcance de cualquier persona. Además ayuda a que la integración de la tecnología con el entorno sea mucho mayor ya que es más fácil no tener en cuenta toda la tecnología que nos rodea si no tenemos que interactuar físicamente con ella.

Siguiendo con el ejemplo de la casa domótica, es mucho más agradable para el usuario llegar a casa y bajar las persianas con un gesto o encender las luces con un comando verbal que tener que usar su teléfono o un panel de control instalado en su hogar para realizar estas opciones. Además este tipo de controles permite que un mayor número de usuarios pueda hacer uso de la tecnología ya que facilita su uso a personas de avanzada edad, o usuarios con algún tipo de discapacidad.

Actualmente ya hay disponibles en el mercado dispositivos que permiten realizar tareas como poner música, establecer alarmas o controlar las luces sin necesidad de interactuar físicamente con el dispositivo ya que todas las operaciones se realizan mediante órdenes por voz.

Ilustración 5: Imagen promocional de Amazon Echo. Fuente:Amazon

2.1.4 Computación Ubicua

El concepto de inteligencia ambiental está íntimamente ligado al de computación ubicua. Este concepto hace referencia a la capacidad de los sistemas de desaparecer, convirtiéndose en “una parte invisible e integral de la vida de las personas” (Mark Weiser,1991).

Para conseguir esto, es necesario que los dispositivos suficientemente integrados en el entorno y que la interacción con los mismo se realice de una manera natural para el ser humano. Además los usuarios necesitan poder acceder a la información desde cualquier lugar lo que es posible hoy en día gracias a los múltiples dispositivos móviles de los que disponemos.

Según Mark Weiser (1991), los dispositivos de computación deben ser algún día como las señales de tráfico, vayas de publicidad o incluso envoltorios de caramelos. Todos estos elementos forman parte de nuestro entorno cotidiano y contienen información que podemos consultar en cualquier momento sin ni siquiera prestar atención.

Como podemos observar, todos estos conceptos están muy relacionados con la idea de la Inteligencia Ambiental. Esta persigue el objetivo de ser ubicua, pero va más allá, las aplicaciones de Inteligencia Ambiental no buscan únicamente ofrecer información al usuario sino ofrecer servicios personalizados que hagan la vida más fácil y que sean capaces de adaptarse a los cambios tanto en el entorno como en los propios usuarios.

Para conseguir estos objetivos, la computación ubicua es, sin lugar a dudas, una tecnología imprescindible.

2.1.5 *Internet of Things*

Ilustración 6: Representación del IOT

El Internet de las cosas (*Internet of Things*) o simplemente IoT es, otro de los conceptos íntimamente ligados al de Inteligencia Ambiental. El Internet de las cosas, se define como un sistema de dispositivos de computación interrelacionados, conectados entre sí a través de algún tipo de red de comunicaciones y que tienen la capacidad de transmitir información sin necesitar de la intervención de un humano.

La idea es que cualquier dispositivo que disponga de sensores y, lo que es más importante, sea capaz de transmitir la información que recoge. De esta forma podemos recolectar ingentes cantidades de datos en tiempo real para procesarlos posteriormente y utilizarlos según nos convenga, por ejemplo facilitándonos la toma de decisiones ante problemas complejos.

Como ejemplo de un proyecto IoT, vamos a comentar brevemente el proyecto VLCi, una plataforma para ciudades inteligentes desplegada en la ciudad de Valencia. El proyecto comenzó en 2014 con el objetivo de aportar a la ciudad soluciones en cinco áreas: movilidad, gobernanza, entrono, sociedad y bienestar.

“En los últimos 4 años, entre otras actuaciones, el proyecto de plataforma de ciudad inteligente o proyecto VLCi, ha conseguido romper los silos de información existentes en cada uno de los servicios verticales del Ayuntamiento, pudiendo así construir recursos informáticos orientados a una mejor gestión municipal pero también a dar al ciudadano toda la información a través de múltiples recursos

que se han puesto a su disposición: el App Valencia para móviles tanto Android como iOS, el portal de transparencia y datos abiertos, el geoportal, el portal "valenciaalminut" y el cuadro de mando de ciudad para el Ayuntamiento." (ESMARTCITY, 2018).

Desde la web de la plataforma http://smartcity.valencia.es/es/ es posible acceder a sus múltiples servicios como por ejemplo:

- València al Minut: Proporciona a la ciudadanía un portal de información para conocer en tiempo real el estado de la ciudad, entre otros datos, se muestra información sobre el transporte público, niveles de contaminación del aire, información meteorológica, etcétera.

- Geoportal: Pone a disposición de la ciudadanía una gran variedad de recursos existentes del Ayuntamiento de Valencia.

  El Geoportal ofrece información proporcionada a través de una vista de mapas por diferentes áreas municipales como movilidad, economía, servicios sociales, cultura festiva, etc. en la que el usuario puede seleccionar la información de interés a través de un mapa y elegir las que desee visualizar.

- Cuadros de Mando: Este servicio ofrece a los responsables públicos información pormenorizada y actualizada sobre el estado de los diferentes servicios públicos de la ciudad con el objetivo de facilitar la toma de decisiones y ayudar a conseguir una gestión más eficiente.

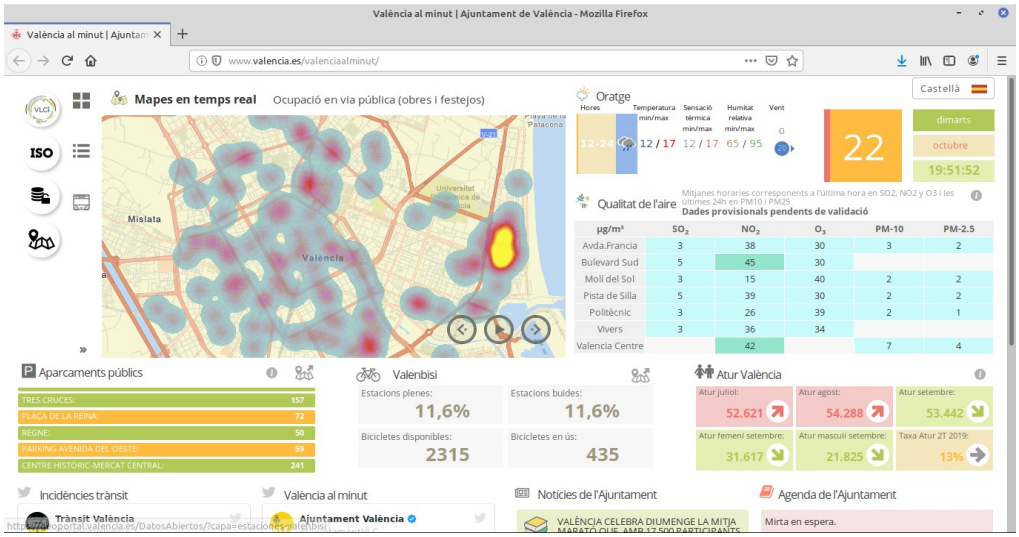

Ilustración 7: Portal València al minut. Fuente: www.valencia.es/valenciaalminut/

2.2 Tecnologías Empleadas

2.2.1 *JSON*

JSON (acrónimo de *JavaScript Object Notation*, "notación de objeto de *JavaScript*") es un formato de texto ligero e independiente del lenguaje de programación para el intercambio de datos de forma que estos sean fácilmente legibles para las personas y a la vez fáciles de interpretar por un ordenador.

Fue especificado por Douglas Crockford a comienzos de la década de los dos mil siendo finalmente estandarizado en dos mil trece como ECMA-404. Actualmente su uso está ampliamente extendido, como veremos en esta sección tanto *Owntracks* como *MQTT* utilizan este formato a la hora de enviar y recibir mensajes.

2.2.1.1 Estructura de un mensaje *JSON*

A lo largo de este documento se describirá la estructura y contenido de varios mensajes que emplean el formato *JSON*, por ello resulta conveniente explicar brevemente su estructura a fin de facilitar su comprensión.

Un mensaje escrito siguiendo el formato *JSON* está construido alrededor de dos estructuras, una colección de pares clave/valor y una lista ordenada de valores.

Un ejemplo de mensaje *JSON* para describir a una persona puede ser el siguiente:

```
{
  "nombre":"Joan",
  "apellido1":"Lluis",
  "apellido2":"Vives",
  "edad":"55",
  "telefonos":["111222333","444555666"]
}
```

Como vemos, el mensaje contiene las propiedades que definen a una persona usando *pares clave/valor* siendo la clave el nombre de la propiedad y su valor el valor de la misma. Además de tipos de datos simples como números, cadenas de caracteres o valores booleanos, *JSON* también soporta lista de valores las cuales representan varios valores entre corchetes "[]".

2.2.2 *MQTT*

MQTT (*Message Queuing Telemetry Transport*) es un protocolo de comunicación, diseñado sobre el paradigma publicador-subscriptor,es ligero, simple, abierto y diseñado para ser fácil de implementar. Estas características lo hacen ideal para ser usado en poder utilizarse en entornos con poco ancho de banda o en situaciones en las que se quiere minimizar el uso de la misma. (International Organization for Standardization [ISO], 2016)

Fue desarrollado por IBM junto con Arcom en 1999, desde entonces ha sido utilizado extensivamente en múltiples sectores principalmente a raíz de la irrupción del IoT tanto en el sector industrial a la hora de automatizar y monitorizar los procesos de producción como en el sector servicios prestando soluciones de monitorización y control de pequeños dispositivos integrados generalmente en amplias redes de aparatos interconectados. Algunos de los proyectos en los que se ha hecho uso de *MQTT* son:

1. Microsoft Azure IoT Hub
2. Ejabberd
3. McAfee OpenDXL

Al tratarse de un protocolo que sigue una arquitectura de publicadores y subscriptores, es necesario dispones de uno o varios mediadores o *brokers* en nuestros desarrollos basados en *MQTT*, este mediador es típicamente un servidor en el que se está ejecutando un software que haciendo uso de *MQTT* es capaz de recibir y distribuir mensajes.

Los publicadores, como su nombre indica, son los encargados de generar los mensajes. En este tipo de protocolos, a diferencia de otros patrones de comunicación los mensajes no tienen un destinatario específico, es decir, en el momento de publicar un mensaje un publicador no puede especificar a quién quiere que le llegue el mensaje. Lo que hace el publicador es publicar este mensaje en un tema o *topic.* Estos temas se pueden entender como un canal de comunicación al que los distintos usuarios pueden suscribirse para recibir los mensajes que se publiquen en el canal.

Ilustración 8: Ejemplo de un topic. Fuente: HiveMQ

Como vemos en la figura anterior, los temas pueden estructurarse en múltiples niveles, esto nos permite crear una estructura en forma de árbol para poder organizar nuestros mensajes de la mejor forma posible.

Cuando un publicador genera un mensaje en un determinado tema, este mensaje se envía al mediador o *broker* que a su vez propagará el mensaje a todo aquel que esté suscrito al tema en el que se publicó el mensaje.

2.2.3 Owntracks

Owntracks es una aplicación que ofrece servicios de posicionamiento y localización GPS. Se trata de una aplicación gratuita y de código abierto desarrollada originalmente para Android por Alexander Rust, Christoph Krey y Jan-Piet Mens. La aplicación permite a los usuarios almacenar su posición actual, así como conocer la posición de familiares y amigos (previa autorización de estos) además de definir áreas alrededor de un punto geográfico y detectar cuando un usuario entra o sale de de ellas.

Owntracks también nos ofrece la posibilidad de conectar la aplicación con un broker *MQTT* para poder enviar y recibir mensajes con los cambios de posición que realicen los usuarios de la aplicación o cuando se detecte que han entrado o salido de un área.

Owntracks también nos permite configurar diferentes modos de motorización que van desde un monitoreo continuo de forma automática hasta la ausencia de motorización delegando en el usuario la responsabilidad de actualizar periódicamente los datos sobre su posición.

En cuanto a la conexión con el *broker MQTT*, *Owntracks* nos ofrece conexión segura mediante el empleo de *TLS* (acrónimo de *Transport Layer Security*) el cual es un protocolo criptográfico que proporciona comunicaciones seguras a través de Internet permitiendo la comunicación cliente/servidor diseñado de manera que se evitan escuchas, manipulación o falsificación de mensajes (Internet Engineering Task Force [IETF], 2008)

Mediante este protocolo es posible conectar la aplicación de forma segura con un servidor *MQTT* de modo que nuestros datos de localización estén a salvo de posibles robos o accesos malintencionados a los mismos.

2.2.4 Android y Java

Esta solución ha sido desarrollada usando Java como lenguaje de programación desarrollando la aplicación del servidor y la aplicación móvil con este lenguaje. La aplicación móvil se ha desarrollado para el sistema operativo Android versión siete en adelante.

Android es el sistema operativo más utilizado a nivel mundial, cuenta con más de dos mil millones de usuarios según lo anunciado por Google en dos mil diecinueve. Esto supone el ochenta y seis por ciento de la cuota de mercado en dispositivos móviles (International Data Corporation [IDC], 2020).

Se ha decidido desarrollar la aplicación móvil para Android por tener una ingente base de usuario que nos permite hacer que nuestra aplicación llegue a mucha más gente. Por otro lado la familiaridad con el entorno de desarrollo y el uso de Java como lenguaje de programación de la aplicación han sido factores importantes a la hora de elegir este sistema operativo.

Java por su parte, es un lenguaje de programación orientado a objetos, fuertemente tipado ,imperativo y multiplataforma desarrollado originalmente por James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan para la compañía Sun Microsystems. Lanzado al mercado en mil novecientos noventa y cinco, desde entonces Java se ha convertido en uno de los lenguajes de programación más populares y demandados del mundo con más de nueve millones de desarrolladores utilizando Java y siendo ejecutado en el noventa y siete por ciento de los escritorios empresariales (Oracle, 2020).

Las principales características de java son las siguientes:

Java se ejecuta sobre una maquina virtual por lo que es independiente del *hardware* del sistema en el que se está ejecutando. Los programas escritos en java no se ejecutan directamente en la máquina en la cual están instalados sino que lo hacen sobre una máquina virtual. Existen máquinas virtuales Java para los principales sistemas operativos como Windows o Linux de manera que podemos ejecutar un programa Java en casi cualquier dispositivo independientemente de su arquitectura y sistema operativo subyacente.

La gestión de la memoria de forma de forma automática es otra de las características principales de Java. Al contrario que lenguajes como C o C++, los cuales obligan a los desarrolladores a gestionar manualmente el uso que su programa hace de la memoria del sistema, Java automatiza este proceso y no requiere ningún tipo de gestión activa por parte del programado. Esto no solo repercute en una mayor facilidad de aprendizaje y desarrollo sino elimina la posibilidad de realizar errores en la gestión de la memoria por parte de los desarrolladores ya que esta gestión es, según Oracle, una "fuente fructífera de errores, bloqueos, pérdidas de memoria y bajo rendimiento".

Por último, destacar que Java es un lenguaje orientado a objetos, este tipo de lenguajes de programación se caracteriza por el uso de variables llamadas objetos que contienen tanto datos como instrucciones y que son capaces de interactuar con otros objetos a través de mensajes.

2.2.4.1 Programación orientada a objetos

La programación orientada a objetos supuso un gran cambio en la forma de diseñar y desarrollar software. Stephen (1997) afirmaba: "La programación orientada (POO) a objetos es a la vez evolutiva y revolucionaria. Es evolutiva porque se basa y expande las ideas de la programación estructurada. La POO es revolucionaria porque proporciona algunas características completamente nuevas".

Estas características revolucionarias en su día fueron las siguientes:

- Encapsulación: Los datos y las instrucciones de las variables son tratadas como una sola unidad. Estas unidades son los **objetos** y a la agrupación de datos e instrucciones que sirve de platilla para crear los objetos se la conoce como **clase**.
- Herencia: La posibilidad crear nuevos objetos que extiendan o mejoren a objetos existentes. La herencia nos permite reutilizar código ya existente lo que además de facilitar el desarrollo previene posibles errores y hace que el software sea más fácil de mantener.
- Polimorfismo: Objetos diferentes responden de forma distinta a mensajes iguales. El polimorfismo nos permite realizar una misma acción de varias formas diferentes. Por ejemplo podemos tener un objeto cuya funcionalidad sea realizar operación aritmética de la suma. Gracias al polimorfismo podemos crear una función (método en Java) de nombre

> "Suma" que sume dos valores y al mismo tiempo podemos crear otra función con le mismo nombre pero que use un número distinto de valores.

El polimorfismo nos permite reducir el acoplamiento, es decir como de relacionados están dos clases de nuestro programa. Un alto grado e acoplamiento producirá que al realizar cambios en un componente debamos modificar otros componentes de nuestro software. Por el contrario un software con poco acoplamiento hará que sea mucho más sencillo realizar modificaciones y nos permitirá realizar el mantenimiento de forma más eficaz y con menor probabilidad de introducir errores.

A lo largo de este documento se van a emplear de forma recurrente conceptos relacionados con la programación orientada a objetos a la hora de explicar la implementación de la solución propuesta.

2.2.4.2 Librería Log4J 2

Para poder disponer en la aplicación del servidor de una funcionalidad que permita llevar el registro (*log*) de todo lo que ocurra durante la ejecución de la aplicación, se ha decidido emplear la librería Log4J en su versión 2. Esta librería es un solución popular para añadir trazabilidad en aplicaciones desarrolladas en Java, se distribuye bajo la licencia *Apache Software License* la cual es una licencia de código abierto.

"*log4j* está diseñado para ser fiable, rápido y extensible. Dado que el registro rara vez es el enfoque principal de una aplicación, la API de *log4j* se esfuerza por ser simple de entender y usar" (Welcome to Log4j 2!, 2021)

Usar esta librería nos permite, de una manera sencilla poder generar ficheros de texto donde queden registrados los posibles errores que se produzcan durante la ejecución de la aplicación del servidor de forma que resulte mucho más sencillo identificar la causa de estos errores y poder realizar el mantenimiento que sea necesario a la aplicación.

2.2.5 *SQLite*

De entre las múltiples tecnologías de base de datos que hay disponibles, se ha optado por emplear en este proyecto una base de datos relacional, en concreto, se ha utilizado *SQLite*. Se trata de un motor de base de datos rápido, de reducido tamaño, auto-contenido, con una gran fiabilidad y que nos ofrece todas las funcionalidades que se esperan de este tipo de herramientas. *SQLite* es ligero en lo que respecta a la complejidad de la configuración, la sobrecarga administrativa y el uso de recursos (Kreibich, Jay A., 2010).

Por otra parte, sus creadores han decidido que tanto el código como la documentación sean de dominio público por lo que son accesibles por todo aquel que lo desee. Frente a otras alternativas que requieren de la compra de una licencia para su uso, *SQLite* se presenta como una solución ideal para un proyecto como este.

Una de las características fundamentales por las que se ha decido usar *SQLite* ha sido el hecho de que no necesita un servidor para funcionar. Muchas de las alternativas existentes necesitan de un proceso ejecutándose continuamente para poder funcionar, este proceso es el que se encarga de ejecutar las diferentes operaciones sobre la base de datos. Al no necesitar de este proceso, no es necesario configurar y mantener un servidor adicional que solo haría que añadir una complejidad innecesaria a la solución presentada.

A pesar de estas ventajas, hay situaciones en las que es mejo emplear otro tipo de tecnologías, por ejemplo el tamaño de una base de datos *SQLite* está limitado a ciento cuarenta *terabytes*, más que de sobra para este proyecto, por lo que se deberá emplear otras tecnologías a la hora de manejar grandes cantidades de información.

Otra es que su rendimiento no está optimizado al utilizarse en aplicaciones cliente-servidor, donde un gran número de clientes realizan peticiones a la base de datos. Debido a como funciona internamente *SQLite*, que se ve afectado por las latencias de los sistemas de ficheros. Como regla general se recomienda evitar su uso en aplicaciones donde la base de datos sea utilizada directamente y de forma simultánea por multitud de dispositivos a través de la red.

2.2.6 Otras tecnologías consideradas

Además de las tecnologías comentadas anteriormente, se consideró el uso de otras que finalmente fueron descartadas.

Se consideró la posibilidad de desarrollar nuestra propia solución de geolocalización como parte de la aplicación móvil en lugar de emplear *Owntrasck*. Sin embargo, esta opción fue descartada ya que aumentaría la complejidad de la solución sin aportar grandes beneficios. Por otro lado, *Owntracks* es una aplicación ampliamente usada y probada lo que ofrece una mayor seguridad.

También se consideraron varias opciones para el sistema de base de datos. *MySQL* fue una opción considerada como alternativa a *SQLite*. Sin embargo a diferencia de esta última *MySQL* necesita un servidor para poder ejecutarse, además un servidor *MySQL* tiene un tamaño de seiscientos mega *bytes* mientras que *SQLite* son doscientos cincuenta kilo *bytes*, esta gran diferencia de tamaño puede ser importante si el *hardware* sobre el que se despliega la solución tiene un espacio de almacenamiento muy reducido.

Por otro lado *MySQL* soporta más tipos de datos, es más escalable y es capaz de gestionar mejo grandes cantidades de datos. Sin embargo el volumen de datos en este proyecto no es tan elevado como para justificar el uso de este sistema.

3. Caso de estudio

3.1 Introducción

Como hemos visto, la inteligencia ambiental aspira a mejorar nuestra vida facilitándonos la forma en la que nos relacionamos con la tecnología que nos rodea. Resulta totalmente lógico pensar en las posibles aplicaciones que este tipo de tecnologías pueden tener.

Algunas de las aplicaciones más interesantes son aquellas relacionadas con las distintas actividades económicas. En el sector industrial toma fuerza de la idea de la cuarta revolución industrial o industria 4.0. Esta revolución consistiría en la puesta en marcha de fábricas "inteligentes" capaces, entre otras cosas, de adaptarse más fácilmente a los procesos y necesidades de producción. Se trata de revolucionar la industria aplicando las nuevas tecnologías para obtener más datos sobre los procesos productivos y el estado de las líneas de producción.

Pero no solo en el sector industrial se pueden aplicar la inteligencia ambiental, el sector turístico es uno que, por su idiosincrasia, resulta especialmente adecuado para aplicar tecnologías enfocadas a la inteligencia ambiental.

El reto consiste en aplicar todo este abanico de tecnologías y paradigmas de manera que sea posible desarrollar aplicaciones que aporten valor a la actividad turística sin que supongan un gran coste en términos económico para que lleguen a la mayor cantidad de gente posible.

3.2 La Inteligencia Ambiental Aplicada a la actividad turística

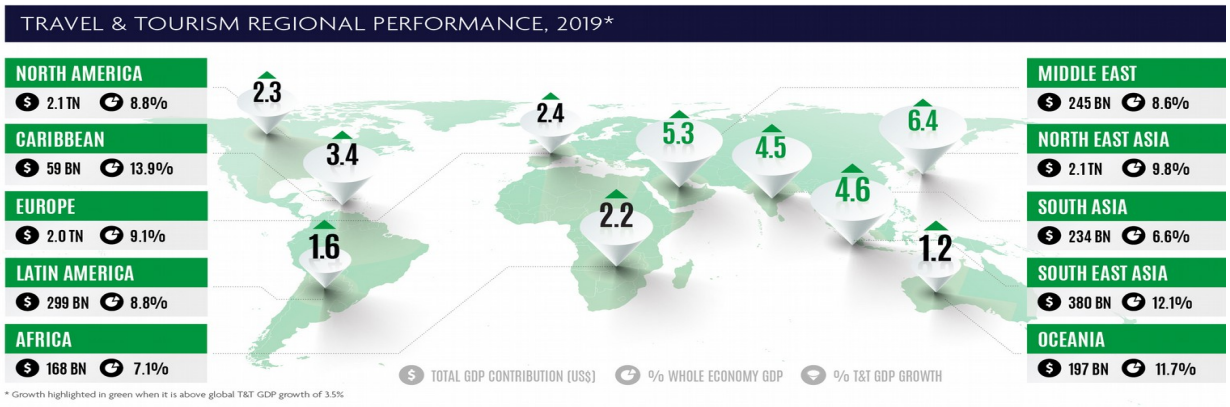

Ilustración 9: Impacto del sector turístico en la economía mundial. Fuente: WTTC.

En el año dos mil dieciocho la actividad turística generó más de un diez por ciento del producto interior bruto mundial, esto es, unos ocho coma ocho trillones de dólares estadounidenses siendo uno de los sectores con mayor peso en la economía global y creando uno de cada diez empleos con un total de trecientos diecinueve millones de personas empleadas en el sector turístico en todo el mundo. (World Travel and Tourism Council, 2019).

En España, la actividad turística es una de las principales actividades económicas de nuestro país la cual aporta un catorce coma seis por ciento del producto interior bruto y un catorce coma siete por ciento del total de empleos (World Travel and Tourism Council, 2019).

España es uno de los destinos más visitados a nivel mundial. Según datos de la Organización Mundial del Turismo, en el año dos mil dieciocho llegaron a nuestro país más de ochenta y ocho millones de turistas extranjeros los cuales gastaron cerca de ochenta y dos mil millones de euros (Organización Mundial del Turismo, 2019) .

La creciente importancia del turismo en la economía global y especialmente en la española, hace que explorar las posibles aplicaciones de nuevas tecnologías como la Inteligencia Ambiental resulte más que interesante de cara a potenciar uno de los grandes motores económicos de país.

Para ello, debemos primero conocer las características de sector turístico español. España recibe cada año más de cincuenta millones de turistas extranjeros, la mayoría se concentran en las grandes ciudades (sobre todo Madrid y Barcelona) así como las Islas Canarias y las Islas Baleares.

Ilustración 10: Impacto del turismo en la economía española. Fuente: WTTC

El perfil de turista varía mucho debido a la gran cantidad de visitantes que recibe el país, así como a la amplia oferta turística que ofrece. Según datos del Instituto Nacional de Estadística, los turistas gastaron de media ciento cuarenta y dos euros por día de estancia en España. Esto nos puede indicar que la mayoría de las turistas que nos visitan no tienen un gran poder adquisitivo. Generalmente, este tipo de turistas prefiere el turismo de bajo coste, el llamado turismos de "sol y playa".

En los últimos años, sector turístico no ha sido ajeno a la proliferación de aplicaciones colaborativas que pretenden ampliar y mejorar la experiencia de los turistas y es ahí donde la inteligencia ambiental puede jugar un mayor rol. La explosión de los servicios colaborativos ha hecho que cambie la concepción de turismo tal y como la conocíamos, cada vez es más fácil reservar vuelos y hoteles al margen de los operadores habituales.

Aplicaciones como *AirB&B*, *skyscanner* entre otras, han conseguido abaratar los precios y ofrecer a sus usuarios una facilidad a la hora de contratar sus servicios difíciles de igualar por parte de los operadores tradicionales.

Sin embargo, partiendo de la base que los turistas que nos visitan buscan en su mayoría actividades de bajo coste, un enfoque consiste en la creación de aplicaciones que ofrecen información sobre las actividades que se pueden realizar en el área, estas aplicaciones se basan en la información que proporcionan los usuarios de forma colaborativa y permiten planificar de una mejor manera los viajes.

En los últimos años, las formas más tradicionales de turismo han ido dejando paso a nuevas formas de visitar ciudades, parajes y otras atracciones. Cada vez más, los turistas se sirven de la tecnología para organizar y disfrutar sus vacaciones; de contratar las vacaciones en una agencia de viajes, hemos pasado a contratar el alojamiento y el vuelo por nuestra cuenta a través de la *web.* En lugar de buscar en guías de restaurantes, ahora lo hacemos a través de una aplicación móvil colaborativa en la que miles de usuarios publican sus valoraciones y experiencias.

En definitiva, la forma de hacer turismo está cambiando y es en este contexto donde ofrecer nuevos y mejores servicios y experiencias a los turistas resulta fundamental para mantenerse competitivos y conseguir seguir atrayendo grandes volúmenes de turistas.

Es por ello que, de la mano de las nuevas tecnologías, empresas e instituciones tratan de innovar en su oferta turística para atraer a un mayor número de clientes y ofrecer experiencias más satisfactorias.

Hoy en día, es común encontrar recursos como páginas web o aplicaciones móviles cuando visitamos algún monumento, mueso u otra atracción turística gracias a las cuales podemos obtener información sobre el lugar que estamos visitando o sobre los objetos que podemos encontrar en el. Sin embargo este tipo de herramientas nos ofrecen la información de forma pasiva, son los usuarios los que deben buscar esa información en lugar de ser el propio servicio el que nos ofrezca esta información según la vayamos necesitando.

La idea es que sea la tecnología la que se adapte a nosotros y no nosotros a ella, para ello necesitamos obtener de forma constante información sobre el entorno, por ejemplo realizando un seguimiento de la posición del usuario.

Algunas de las soluciones que se están implementando hoy en día requieren del uso de una cuantiosa infraestructura ya que emplean en su mayoría dispositivos *Bluetooh* lo que además de de suponer un incremento en los costes debido a la adquisición de estos dispositivos requieren de un mantenimiento más dedicado y costoso.

Por otro lado, otras soluciones apuestan por el uso de códigos *QR* que el usuario debe escanear con su dispositivo móvil. Este tipo de soluciones, a pesar del evidente beneficio de tener unos costes más bajos tienen varias desventajas como por ejemplo que, en caso de que los códigos estén a la intemperie necesitarán un mantenimiento continuo para evitar que se deterioren y queden inservibles. Además, tipo de soluciones requieren que el usuario tome un rol activo escaneando los códigos lo que puede ser complicado para personas mayores que no estén acostumbradas a este tipo de tecnologías e incluso puede llegar a resultar tedioso si los usuarios se ven obligados a tener que estar continuamente escaneando códigos durante un largo periodo de tiempo.

Pero a pesar de estos inconvenientes o retos a resolver, la aplicación de nuevas tecnologías nos ofrecen grandes oportunidades a la hora de mejorar la experiencia de los usuarios en el sector de la actividad turística. El hecho de poder ofrecer contenidos en base a la posición del usuario nos permite ofrecer una experiencia personaliza para cada usuario, mostrándole información a medida que avanza en su visita de manera que cada uno pueda disfrutarla a su ritmo.

Nuestro objetivo debe ser emplear las tecnologías disponibles para conseguir un sistema que nos permita proporcionar esta información en base a la posición de los usuarios sin que suponga un coste demasiado elevado, es decir, debemos buscar una solución de bajo coste que permita que este tipo de tecnologías lleguen al mayor número de personas posibles.

3.3 El castillo de Sagunto

Teniendo en cuenta todo esto, se ha decido estudiar el caso del castillo romano de Sagunto y la posible aplicación de un sistema de geolocalización e interacción con los diferentes monumentos y espacios que componen este monumento nacional, una de las principales atracciones del municipio.

El castillo de Sagunto, situado en el municipio homónimo es una edificación defensiva construida alrededor del siglo XI sobre los restos de los anteriores asentamientos romanos e íberos. Se encuentra dividido en varias plazas y rodeado por una muralla en la que se entremezclan trozos de diferentes eras. Construido sobre un cerro, el castillo domina la población de Sagunto y sus alrededores. En su interior, nos podemos encontrar con restos de las diferentes culturas que a lo largo de los siglos han hecho uso de su estructura convirtiendo al castillo de Sagunto en un monumento arqueológico de gran relevancia como así lo atestigua su declaración como monumento nacional en el año mil novecientos treinta y uno.

La problemática a la que nos enfrentamos en este caso es conseguir geolocalizar varios de los espacios que componen esta fortaleza así como los recursos arqueológicos más relevantes de la misma para poder ofrecer a los visitantes información geolocalizada que les permita conocer la historia del castillo mejorando la experiencia de su visita.

Actualmente el castillo se encuentra en un estado próximo al abandono donde la información que se ofrece a los visitantes es escasa e incluso inexistente en algunas partes del monumento como por ejemplo en el centro de visitante que llevan años abandonado debido a la falta de corriente eléctrica. Esta situación hace necesaria nos solo aumentar la cantidad de información que se ofrece a los visitantes sino también ofrecerla de una forma más accesible y cómoda para ellos.

La solución desarrollada ofrecerá a los visitantes del castillo información sobre las diferentes zonas y los diversos restos arqueológicos de forma automática empleando una aplicación móvil. Se tendrá en cuenta la posición del visitante para poder ofrecerle información de los recursos que tenga más próximos de forma que, según se vaya desplazando por el castillo recibirá en su teléfono móvil esta información de forma automática pudiendo además consultar sobe un mapa la posición de los diversos recursos que tenga en las cercanías.

Hay que tener en cuenta que la infraestructura con la que cuenta el castillo es prácticamente inexistente por lo que la solución propuesta debe ser capaz de funcionar sin que sea necesario un gran despliegue de tecnología en el castillo.

Por lo tanto, podemos observar que la solución a desarrollar debe solucionar los siguientes problemas:

- Ser capaz de localizar la posición del usuario en tiempo real según se desplaza por el castillo de Sagunto.
- Poder geolocalizar los diferentes elementos (espacios abiertos y restos arqueológicos) que conforman el monumento del castillo.
- Ofrecer información de estos recursos geolocalizados según la proximidad del usuario a los mismos.
- Emplear la menor infraestructura posible para implementar la solución evitando el empleo de dispositivos de geolocalización (balizas).

Una vez hemos visto el contexto del castillo de Sagunto y los problemas que debemos resolver debemos seleccionar los recursos que van a ser geolocalizados al desarrollar la solución. Para ello, se van a seleccionar las áreas más importantes del monumento como son como son la plaza de armas, la plaza de la almenara y la plaza de los nueve pilares. Además en la plaza de armas, en la que se encuentran la mayoría de los restos arqueológicos, se van a geolocalizar los siguientes elementos: los restos del foro, el aljibe, y la puerta de la almenara.

Por último se va a incluir la entrada como un área geolocalizada más para poder ofrecer información a los visitantes antes de comenzar la visita.

Una vez definidos los distintos elementos que vamos a geolocalizar, debemos obtener las coordenadas geográficas de los mismos. Para ello podemos emplear nuestro teléfono móvil, si este dispone de GPS, o herramientas como Google Maps. En este caso el resultado es el siguiente:

- **Entrada al castillo**: Latitud: 39.675972, Longitud:-0.277447
- **Plaza de armas**: Latitud: 39.675943, Longitud: -0.276477
    - Restos del Foro: Longitud 39.675957, Latitud: -0.276516
    - Aljibe: Longitud:39.675681, Latitud: -0.276451
- **Plaza de la almenara**: Latitud: 39.676439, Longitud: -0.275645
    - Puerta de la almenara: Latitud: 39.676199, Longitud: -0.276290
- **Plaza de los nueve pilares**: Latitud: 39.676776, Longitud: -0.276553

4. Análisis del problema

Partiendo del caso de estudio que hemos visto en el apartado 3 podemos concluir que nos enfrentamos a varios problemas que hemos de resolver para poder implementar una plataforma que permita la interacción entre los visitantes del castillo de Sagunto y los restos arqueológicos que forman parte del castillo.

Primero debemos discutir que entendemos por geolocalizar un determinado elemento, la idea es obtener la posición geográfica de este elemento la cual vendrá definida por la latitud y la longitud de un punto sobre la superficie terrestre. De esta manera seremos capaces de relaciones esa posición con el recurso que deseamos geolocalizar.

En el caso de que el objeto que deseamos geolocalizar esté estático podemos emplear un dispositivo que cuente con un sistema de posicionamiento global o *GPS* para obtener las coordenadas que nos interesen aunque también podemos recurrir a otro tipo de herramientas como *Google Maps* o similares para obtener estos datos. Si por el contrario el objeto puede cambiar de posición necesitaremos dispositivos capaces no solo de registrar la posición actual en la que se encuentre el objeto a geolocalizar sino también de transmitir esta información para actualizar constantemente la posición del recurso geolocalizado.

Para esta última tarea dispositivos como los teléfonos móviles actuales pueden ser una buena opción debido a su uso extendido aunque también se puede recurrir a otro tipo de dispositivos como localizadores o balizas *GPS* algunas de las cuales cuentan incluso con paneles solares para garantizar el suministro de energía incluso si no se puede disponer de una infraestructura eléctrica.

Una vez hemos visto en que consiste y como podemos conseguir geolocalizar algo o a alguien vamos a definir los objetos que se van a geolocalizar. Estos pueden ser de dos tipos, por un lado tenemos los **recursos geolocalizados** que son elementos, objetos, bienes, estructuras, etc. que se pueden geolocalizar con un único punto. Por ejemplo, para el castillo de Sagunto se han seleccionado varios recursos como por ejemplo: los restos del foro o la puerta de la almenara.

Por otro lado también podemos geolocalizar espacios o zonas que nos es imposible localizarlas únicamente usando un punto geográfico. En este caso estamos hablando de **áreas geolocalizadas** las cuales las podemos geolocalizar empleando su centro y un radio. De este modo podemos considerar que cualquier punto que se encuentre a menos distancia que su radio del punto central estará dentro del área geolocalizada lo que nos permite geolocalizar grandes espacios de una forma sencilla.

Cada uno de los recursos y áreas geolocalizadas llevarán asociados ciertos datos como su posición definida mediante la latitud y la longitud tendrá, su nombre, una imagen descriptiva y la información sobre el recurso o el área que puede ser de interés para los usuarios.

4.1 Especificación conceptual

Para modelar esta información se ha planteado el siguiente modelo de datos:

Las principales entidades con las que trabaja la solución son los recursos y los eventos. Los recursos serán tanto áreas como recursos geolocalizados y tendrán toda la información necesaria incluyendo sus coordenadas geográficas.

Los eventos representan eventos de entrada o salida de un área geolocalizada. Tiene la información sobre el área en la cual se ha generado el evento así como el tipo del mismo pudiendo ser de entrada cuando un usuario entra en un área o de salida en el caso contrario.

En cuanto a la información que puede tener un recurso o un área geolocalizada, esta se divide en varias partes. Por un lado está la información sobre la geolocalización del recurso se almacena en un objeto llamado *PuntoGeografico* el cual almacenará la latitud y la longitud en la que se encuentra ese recurso o donde se localiza el centro de un área geolocalizada.

Por otro lado tendremos objetos de tipo *RecursoGeolocalizado* o *AreaGeolocalizada* los cuales tendrán información como nombre, descripción y una o varias fotografías. Se ha decido que en lugar de almacenar las imágenes en el servidor y posteriormente enviarlas a los usuarios cuando junto con los datos de los recursos, se usaría una ruta o *URL* que identificaría la imagen y permitiría a la aplicación móvil descargarla directamente, de esta manera conseguimos reducir el tamaño de los mensajes que ha de manejar la solución y conseguimos que su envío sea más ágil.

Un recurso o área puede tener más de una imagen asociada, por un lado podemos asignarle una imagen principal que debería servir para representar a ese recurso o área y, por otra parte se pueden asociar múltiples imágenes extra para que a la hora de proporcionar información sobre el área o el recurso geolocalizado, los usuarios dispongan de más fotografías.

Veamos con un ejemplo el uso de este modelo de datos. Usemos el rectorado de la Universidad Politécnica de Valencia está situado en las siguientes coordenadas:

- Latitud: 39.4817218
- Longitud: -0.345629

Para no tener que indicar el punto cardinal, el Norte y el Este se representan con valores positivos, por el contrario, el Sur y el Oeste se representan con valores negativos.

Siguiendo con el ejemplo del rectorado, podemos definir el recurso de la utilizando un objeto de tipo *RecursoGeolocalizado* en el cual tendremos por un lado la información del recurso:

- *nombre*: Rectorado de la Universidad Politécnica de Valencia.
- *urlImagen*: http://www.upv.es/plano/imagenes/3AG.jpg
- *info*: El rectorado de la UPV tiene su sede en el edificio 3A, situado en el campus de Vera en Valencia. Teléfono: +34 963877405.

Y por otro lado, tendremos la información de la posición del recurso geolocalizado.

- Punto Geográfico:
  - *latitud*: 39.4817218
  - *longitud*: -0.3446425,17

De esta manera tendríamos definido un recurso, el rectorado, en base a su posición y dispondríamos de la información relacionada con el mismo.

De la misma manera que definimos recursos podemos definir también áreas geolocalizadas en las que, cuando un usuario entre en ellas se generarán eventos que indicarán si el usuario ha entrado o salido del área. La información de este evento se guarda empleando un objeto de tipo *EventoArea* que estará compuesto por la siguiente información:

- *tipoEvento*: Representa el tipo de evento generado, este puede ser de entrada en un área o de salida.
- *nombreWaypoint*: Nombre dado al área con la cual ha interactuado el usuario.
- *idWaypoint*: Identificador del área con la cual ha interactuado el usuario.

4.2 Especificación abstracta de mensajes

A la hora de transmitir los datos usando el modelo que hemos definido en el apartado anterior se emplearán varios tipos de mensajes diferentes.

El primero de ellos se utilizará para enviar la información sobre la posición de un usuario cuando este se mueva, este mensaje deberá incluir el nombre del usuario, la latitud, la longitud de la coordenada geográfica en la que se encuentre y una lista con los identificadores de las áreas geolocalizadas dentro de las cuales esté el usuario.

El segundo de los mensajes se utilizará para enviar información sobre un conjunto de recursos o áreas geolocalizadas entre los componentes de la solución. Este mensaje estará formado por el nombre del usuario destinatario del mensaje y una colección de datos sobre los recursos. Estos datos serán: el nombre del recurso, su descripción, la URL desde la que descargar una fotografía, así como la latitud y la longitud de la posición del recurso o del área geolocalizada.

También tendremos mensajes que se emplearán cuando un usuario entre o salga de un área, estos mensajes estarán formador por el nombre del área, el nombre del usuario y el tipo de evento (entrada o salida). Cuando se produzca una entrada o una salida de un área geolocalizada será necesario poder transmitir la información relacionada con ese área. Para ello se empleará un mensaje que estará compuesto por el nombre del área, su descripción y su fotografía.

Por último, cuando un usuario entre por primera vez a la aplicación se producen dos mensajes. Por un lado se genera un mensaje que contiene el nombre del nuevo usuario y, cuando este mensaje llega al servidor, se genera otro mensaje que contendrá información sobre las diferentes áreas geolocalizadas que existan en ese momento en el sistema. La información de cada una de estas áreas será: tipo, descripción, latitud, longitud, radio, *id, timestamp.*

5. Diseño de la solución

El diseño que se plantea consiste en una solución componente en la cual tendremos cuatro elementos claramente diferenciados: un servidor encargado de procesar la información sobre la posición de los diferentes usuarios, una aplicación móvil en la que el usuario podrá consultar los datos de los recursos geolocalizados cercano, una aplicación de geolocalización que registrará la posición de los usuarios y transmitirá esta información al servidor y, por último un *broker* que se encargará de gestionar los mensajes entre los componentes de la solución.

5.1 Arquitectura de la solución

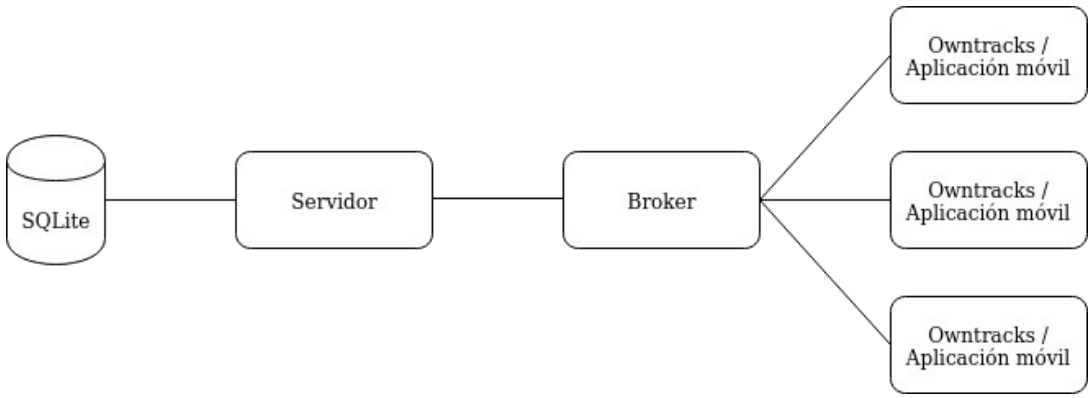

Ilustración 11: Diagrama de los principales componentes de la solución. Fuente:Elaboración propia.

Como podemos observar en la ilustración anterior, la solución está compuesta por cuatro componentes: el servidor, una base de datos, un gestor de comunicaciones o *broker* y las aplicaciones móviles. El servidor y las aplicaciones móviles no conocen nada el uno del otro ya que la comunicación entre ambos componentes se lleva acabo a través del *broker*, esto implica que es mucho más fácil realizar cambios en estos componentes gracias a que están poco relacionados entre si, es decir tienen un nivel de acoplamiento muy bajo.

El servidor está desarrollado en JAVA y utilizará una base de datos relacional *SQLite*. En cuanto al *broker* o gestor de comunicaciones se ha decidido emplear una solución en la nube, en este caso se ha empleado los servicios proporcionados por *cloudmqtt* que ofrece servicios *brokers MQTT* configurables por el usuario, en nuestro caso particular se ha optado por la versión gratuita por

ser más que suficiente para llevar a cabo el desarrollo de la solución. También se pude optar por otras soluciones como *Mosquitto*, en este caso deberá ser instalado y configurado por los usuarios.

En un momento dado podríamos sustituir por completo el servidor o la aplicación móvil sin tener que realizar cambios en el resto de la solución.

Aunque esta solución puede presentar problemas debido a la ausencia de redundancia de componentes clave como el gestor de comunicaciones, esta aproximación nos ofrece una gran flexibilidad para adaptarla a diversos escenarios sin que sea necesario llevar a cabo grandes modificaciones. Además, de esta manera conseguimos que la infraestructura necesaria sea la mínima posible de manera que se consigan unos coste bajos.

Veamos ahora la función de cada uno de los componentes que forman la solución y como se relacionan entre sí para proporcionar el servicio deseado.

5.1.1 Base de datos y servidor

Estos dos componentes forman el *backend* de la solución planteada, la base de datos almacena la información sobre los recursos y áreas geolocalizadas de manera que estén disponibles para que el servidor pueda acceder a esta información y transmitirla a los usuarios finales cuando se acerquen a un recurso geolocalizado o a un área. Para poder detectar estos eventos, el servidor procesa los mensajes de geolocalización que genera la aplicación *Owntracks* que el usuario lleva instalada en su teléfono móvil. Dependiendo del tipo de evento, el servidor accederá a la base de datos para recuperar la información sobre los recursos cercanos o las áreas a las que haya entrado el usuario y enviará esta información a la aplicación móvil que mostrará los datos al usuario.

Veamos ahora el diseño de la base de datos y del servidor:

En primer lugar, la base de datos estará formada por dos tablas, una para las áreas geolocalizadas y otra para guardar la información de los recursos geolocalizados.

La tabla *areas_geolocalizadas* está formada por las siguientes columnas:

- *id:* identificador del área. Columna de tipo *INTEGER* cuyo valor se genera de forma auto incremental. Esta columna no puede contener valores nulos ni repetidos y actúa como clave primaria.

- nombre: nombre del área geolocalizada. Columna de tipo *TEXT*. Esta columna no puede ser nula.
- información: información sobre el área. Columna de tipo *TEXT.*
- *url_imagen: url* donde se aloja la imagen descriptiva de área. Columna de tipo *TEXT.*
- latitud: valor de la latitud del punto central del área: Columna de tipo *REAL*
- longitud: valor de la latitud del punto central del área: Columna de tipo *REAL.*
- radio: valor del radio del área geolocalizada: Columna de tipo real.

Las columnas latitud, longitud y radio no pueden ser nulas.

La tabla *recursos_geolocalizados* está formada por las columnas:

- id: identificador del recurso geolocalizado. Columna de tipo *INTEGER* cuyo valor se genera de forma auto incremental. Esta columna no puede contener valores nulos ni repetidos y actúa como clave primaria.
- nombre: nombre del área geolocalizada. Columna de tipo *TEXT*.
- información: información sobre el área. Columna de tipo *TEXT.*
- *url_imagen: url* donde se aloja la imagen descriptiva de área. Columna de tipo *TEXT.*
- latitud: valor de la latitud del punto central del área: Columna de tipo *REAL*
- longitud: valor de la latitud del punto central del área: Columna de tipo *REAL.*
- idArea: identificador del área dentro de la cual se sitúa el recurso, esta columna, esta columna puede ser nula y hace de clave ajena a la tabla *areas_geolocalizadas.*

La tabla *imagenes* está formada por las columnas:

- id: identificador de la imagen. Columna de tipo *INTEGER* cuyo valor se genera de forma auto incremental. Esta columna no puede contener valores nulos ni repetidos y actúa como clave primaria.
- *url_imagen: url* donde se aloja la imagen. Columna de tipo *TEXT.*
- id_recurso: identificador del recurso geolocalizado al que pertenece esta imagen: Columna de tipo *INTEGER*
- id_area: identificador del área a la que pertenece esta imagen: Columna de tipo *INTEGER.*

Además de estas dos tablas se han creado tres secuencias para poder generar los identificadores de los recursos, las áreas y las imágenes de forma automática ya que estos identificadores se usan internamente para poder identificar únicamente a estos objetos, el usuario no tiene por que preocuparse por ellos.

Para que la solución funcione correctamente es imprescindible que las áreas geolocalizadas tengan exactamente el mismo nombre que el que se haya definido en Owntracks al crearlas desde la aplicación ya que se emplea el nombre a la hora de buscarlas en la base de datos cuando el usuario entra en un área geolocalizada, por lo tanto si los nombre no coinciden no se podrá recuperar de la base de datos la información sobre el área.

En cuanto al servidor, este se ha diseñado con la siguiente arquitectura:

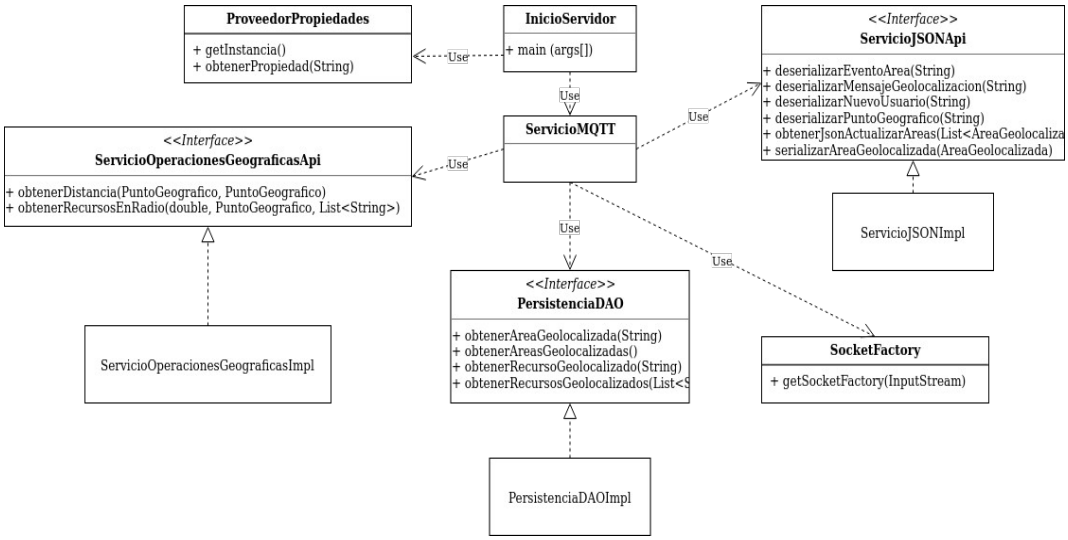

Ilustración 12: Diagrama de los componentes que forman el servidor

Para el diseño de la aplicación del servidor se ha decido emplear un diseño modular formado por varios componentes que nos proporcionan las funcionalidades necesarias para que el servidor sea capaz de llevar a cabo su función. Desde el punto de vista de la arquitectura del servidor, se ha empleado una arquitectura basada en capas, en concreto se ha decidido separar los componentes en tres capas diferentes según su rol en el sistema. De esta manera tendremos una capa de persistencia para el acceso a la base de datos, una capa de lógica de negocio donde se llevarán a cabo las operaciones con recursos y puntos geográficos y una última capa web en la que se encontrará la funcionalidad para comunicar el servidor con las aplicaciones de geolocalización.

Esta arquitectura busca conseguir el menor acoplamiento posible entre los componentes software de la solución para, de esta manera, conseguir una aplicación más fácil de mantener, probar y ampliar gracias a que los cambios que se producen dentro de una capa no afectan a los componentes que se encuentran en capas diferentes. Para conseguir esto, es importante que la comunicación entre las diferentes capas se realice a través de unas interfaces claramente definidas.

Los componentes que forman el servidor son los siguientes:

- *InicioServidor*: Se encarga del arranque del programa, es el responsable de iniciar el resto de módulos.
- *ServicioMQTT*: Se trata del núcleo del servidor. Este componente se encarga de recibir y enviar los mensajes MQTT que se emplean para comunica el servidor con las aplicaciones móviles.
- *ServicioJSON*: Este módulo contiene la funcionalidad necesaria para transformar en objetos JAVA los mensaje que llegan al servidor y viceversa.
- *PersistenciaDAO*: Este componente se encarga de acceder a la base de datos para recuperar información sobre los recursos y las áreas geolocalizadas.
- *ServicioOperacionsGeograficas*: En este módulo reside la funcionalidad necesaria para calcular la distancia entre dos puntos geográficos. Se utiliza para saber que recursos están cerca de la posición indicada por los usuarios.
- *SocketFactory*: Se utiliza para generar *sockets SSL* y de esta forma poder cifrar las comunicaciones entre los clientes y el servidor.

- *ProveedorPropiedades:* Para facilitar la configuración del servidor se ha implementado este servicio que proporciona la funcionalidad para poder consultar valores desde un archivo de texto llamado *config.properties.*

Con esta aproximación modular, la aplicación es mucho más fácil de mantener y modificar ya que la funcionalidad se haya distribuida entre los diversos componentes haciendo que estos servicios sean más pequeños y que su implementación esté mejor estructurada y sea más clara y fácil de leer y comprender.

Por ejemplo, si queremos modificar la forma en la cual los mensajes *JSON* se transforman en objetos Java y viceversa, sabemos que tenemos que modificar la clase de la misma manera que si tenemos que corregir un error a la hora de obtener los datos de la base de datos debemos modificar la clase *PersistenciaDAO*. Y, lo que es más importante, realizar cambios en un modulo no necesita de grandes cambios en el resto de la aplicación. Esta aproximación ayuda a que el software producido sea de una mayor calidad.

Además hacer cualquier cambio en la configuración del servidor como la *URL* del gestor de comunicaciones o la ruta donde se encuentra el archivo de base de datos es una operación trivial ya que solo hay que modificar el archivo de texto *config.properties* y reiniciar el servidor gracias a lo cual la aplicación es más sencilla de mantener y adaptar en el futuro.

Por último se proporciona un servicio de *log* que permite registrar en un archivo de texto diversa información sobre el funcionamiento de la aplicación como las operaciones que se han realizado (suscribirse a un tema, recibir mensajes, publicar en un determinado tema) así como información cuando se produce un error, lo que hace que sea mucho más sencillo averiguar por que ha fallado la aplicación y saber en que estado se encuentra actualmente. En el apartado 6 [pág. 66] se describe en detalle como se usa este servicio en cada uno de los componentes del servidor.

5.1.2 Gestor de comunicaciones

La comunicación de los componentes entre sí se realizará a través de un gestor de comunicaciones también llamado *broker*. Como ya se ha comentado se va a emplear un patrón de comunicaciones publicador-suscriptor de manera que los componentes no se comunican de forma directa si no que en se suscriben a diferentes temas en los que otros componentes publican mensajes.

Para este patrón de comunicaciones es necesario disponer de un gestor que se encargue de controlar los temas existentes y de redirigir los mensajes a cada uno de los suscriptores dependiendo del tema o *topic* en el que se publique.

En este proyecto no se ha desarrollado un gestor de comunicaciones *ad hoc* si no que se ha optado por una de las múltiples opciones disponibles en el mercado, en concreto para desarrollar este prototipo se ha optado por utilizar un gestor de comunicaciones en la nube ya que facilita el desarrollo al no tener que preocuparnos de la instalación ni de su mantenimiento.

5.1.3 Aplicaciones móviles

Los usuarios utilizarán dos aplicaciones móviles, *Owntracks* para comunicar al servidor su posición y la aplicación móvil desarrollada como parte de esta solución para recibir información sobre los recursos cercanos. A pesar de utilizar dos aplicaciones, solo deberá interactuar con la última ya que *Owntracks* no necesita que el usuario realice ninguna acción sobre ella una vez esté configurada y funcionando.

Para la aplicación dedicada a obtener y transmitir la posición del usuario se ha decidido utilizar *Owntracks* ya que, como se ha visto en el apartado dedicado a las tecnologías empleadas, se trata de una solución robusta, segura, probada y gratuita que se ajusta perfectamente a los requerimientos de este proyecto. Los usuarios deberán descargarse la aplicación y realizar una configuración básica para permitir que *Owntracks* pueda comunicarse con el servidor web a través del *broker*. Una vez realizado este paso, la aplicación se ejecutará en segundo plano en el dispositivo de los usuarios enviando información sobre su posición al servidor.

Además, *Owntracks* puede detectar cuando un usuario entre o salga de un región que hayamos definido previamente para hacerlo tenemos dos opciones, o por bien el usuario crea las regiones manualmente en la aplicación o también puede recibir un mensaje en el que se indiquen las regiones que debe tener.

En nuestro caso se ha decidido implementar esta última opción ya que es la cómoda para los usuarios y nos asegura la uniformidad entre todas las aplicaciones al no depender de los usuarios para introducir datos fundamentales para el correcto funcionamiento de la solución.

Por otro lado, deberán instalarse otra aplicación desarrollada en este proyecto en la que podrán consultar un listado con las áreas en las que hayan entrado así como con los recursos geolocalizados que se encuentren cerca de ellos. Este listado se irá actualizando automáticamente según se desplace el usuario sin que este tenga que realizar ninguna acción. Cuando el usuario lo desee puede consultar los datos de un recurso o área concreto para obtener más información sobre el mismo. Además, podrá ver sobre un mapa la posición de los recursos indicando la distancia a los mismos.

Para el diseño de esta aplicación se han seguido los mismos principios que para el diseño del servidor, de manera que se ha optado por una arquitectura en tres capas que nos ayude a compartimentar las diferentes funcionalidades encapsulando estas funcionalidades en componentes con el mínimo acoplamiento entre si.

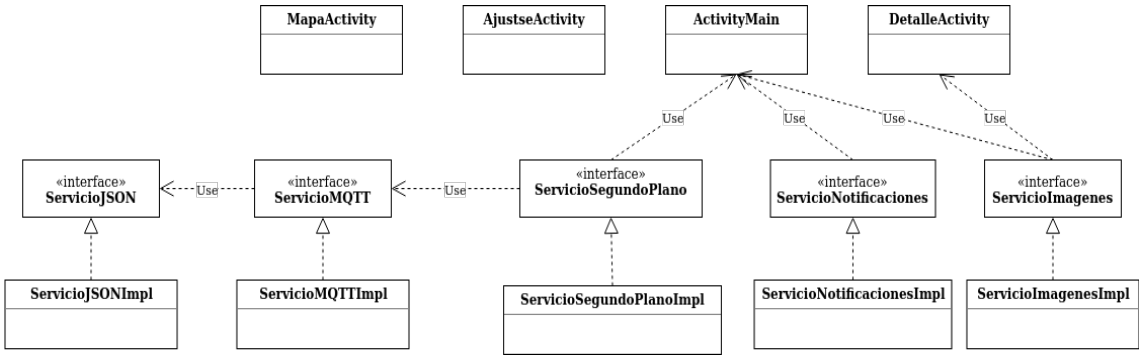

Ilustración 13: Diagrama de los principales componentes de la aplicación

Como se puede observar en el diagrama anterior que describe la arquitectura de la aplicación y los componentes que la forman, podemos distinguir tres capas claramente diferenciadas.

En primer lugar la capa de presentación, la cual se encarga de gestionar todo lo relacionado con la interfaz de usuario. Esta capa es la responsable tanto de "dibujar" la interfaz usando los datos que nos proporcionan el resto de capas como de responder a las acciones que lleve a cabo el usuario. En capa de presentación estarán las actividades y los ficheros *XML*

La segunda capa sería la capa de servicios, en esta capa se encuentran los servicios que controlan la lógica de negocio y nos ofrecen su funcionalidad al resto de capas.

Por último, se encuentra la capa de servicios web que será la encargada de mantener la conectividad entre la aplicación y el servidor utilizando el gestor de comunicaciones. Será en esta capa donde se reciban los mensajes desde el servidor.

Veamos a continuación los diferentes componentes que forman la aplicación móvil.

En la capa de presentación la aplicación consta de tres componentes principales, la actividad *MainActivity* que es la responsable de controlar la ventana principal, la actividad *DetalleActivity* que emplearemos para mostrar información sobre un recurso geolocalizado concreto y la actividad *Mapa*Activity que es la encargada de implementar la funcionalidad necesaria para mostrar recursos geolocalizados sobre un mapa.

También existe una actividad para controlar la ventana de ajustes de la aplicación, donde el usuario puede modificar varios parámetros de la misma. Esta actividad es *AjustesActivity.*

Por su parte, la capa de servicios se compone de cuatro servicios como son el *GestorNotificaciones* y el servicio *ServicioSegundoPlanoMQTT.* El primero se utiliza para poder gestionar las notificaciones que la aplicación muestra a los usuarios al ocurrir ciertos eventos como entrar en un área geolocalizada o al recibir información sobre nuevos recursos geolocalizados cercanos. El segundo de los componentes, *ServicioSegundoPlanoMQTT,* se utiliza para notificar a la actividad principal (*MainActivity*) la recepción de información sobre recursos globalizados de manera que esta pueda actualizar la interfaz de la aplicación de forma acorde a la información recibida.

Por otro lado, debido a que debemos mostrar las imágenes asociadas a los recursos y áreas geolocalizadas, la aplicación dispone de un componente llamado *ServicioImagenes* el cual se encarga de descargar las imágenes de forma asíncrona. Esto último es de vital importancia ya que de no hacerlo así el funcionamiento de la aplicación se vería seriamente afectado.

Otros de los servicios que forma parte de la aplicación es el *ServicioJSON* que funciona de forma análoga al servicio del mismo nombre implementado en la aplicación del servidor aunque con algunas diferencias que veremos más adelante. También existe en la aplicación móvil un componente dedicado a

generar los *sockets SSL* necesarios para la comunicación segura entre la aplicación móvil, el *broker MQTT* y el servidor. De nuevo, este componente funciona de igual manera a la descrita al hablar de la aplicación del servidor [pág. 76].

Por último, la capa de servicios web está formada por el *ServicioMQTT* dedicado a recibir y enviar mensajes desde y hacia el servidor. Aunque la idea detrás de este componente es la misma que la del visto en el servidor [pág. 76], este componente tiene algunas particularidades que destacaremos en próximos apartados.

5.1.3.1 Diseño de la interfaz

Al diseñar el apartado visual y la interfaz de usuario de la aplicación se ha puesto especia cuidado en que la información se presente de una forma clara y accesible de modo que los usuarios tengan que realizar el menor número de interacciones con el dispositivo móvil.

Idealmente, debería ser posible utilizar la aplicación mediante comandos creando de esta manera una forma de interacción mucho más natural y más acorde con los preceptos de la inteligencia ambiental.

Sin embargo esta aproximación requiere de un esfuerzo y de una complejidad que no están al alcance de este proyecto por lo que finalmente, se ha seguido una aproximación más convencional en la que el usuario deberá interactuar físicamente con la aplicación para llevar a cabo ciertas acciones.

Si que se ha tenido en cuenta que el número de interacciones debería ser lo más reducido posible, dejando que sea la aplicación la que automáticamente lleve a cabo ciertas operaciones. Teniendo en cuenta estos conceptos, se han seguido las siguientes pautas a la hora de crear la interfaz gráfica de la aplicación:

- La interfaz debe ser simple para evitar cargar con demasiada información al usuario sin renunciar a mostrar la información necesaria.
- La interfaz debe ser capaz de acomodar automáticamente a distintos tipos, formatos y tamaños de contenido.
- La interfaz debe ser capaz de notificar al usuario los cambios que se produzcan.

Para conseguir estos objetivos se ha diseñado una interfaz de usuario que consta de los siguientes componentes: Una ventana principal de la aplicación que se actualizará con una lista que contenga los recursos geolocalizados cercanos a la

posición del usuario, una ventana en la que se mostrarán esos mismos recursos sobre un mapa y otra ventana que contendrá controles para poder realizar ajustes en el funcionamiento de la aplicación.

Ilustración 14: Ventana de inicio de la aplicación

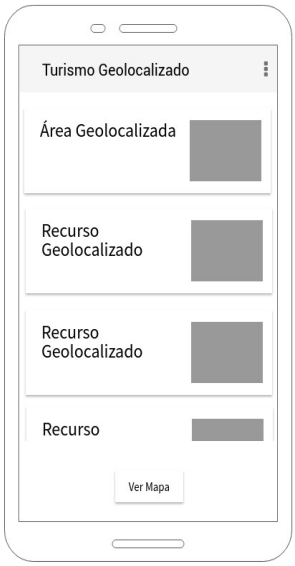

Ilustración 15:Ventana con el listado de recursos geolocalizados cercanos

Las ilustraciones 14 y 15 muestran el diseño de la ventana principal de la aplicación, como ya se ha comentado, es en esta ventana donde aparecen los recursos geolocalizados próximos al usuario. Esta interfaz es también la primera que ve el usuario al utilizar la aplicación. En la ilustración 24 se puede observar el aspecto de la aplicación cuando no se han detectado aún ningún recurso geolocalizado.

En un primer momento se muestra un texto avisando de que no se han detectado recursos para a continuación indicar al usuario que una vez se detecten, estos aparecerán de forma automática en esta ventana. Por último, el usuario puede usar el botón de la parte inferior para consultar la posición de todos los recurso que contiene la base de datos sobre un mapa.

En el momento que la aplicación recibe desde el servidor una lista con recursos y áreas geolocalizadas, la interfaz cambia automáticamente a un formato lista en la que se muestran los recursos indicando su nombre y acompañados de su foto. Para facilitar la distinción entre recursos y áreas. El color de los elementos de la lista cambia según el tipo de elementos que sean. Para los recursos se ha elegido como fondo un color azul mientras que para las áreas se utiliza un tono ocre para que sea más fácil de distinguir los dos tipos de objetos.

Al mismo tiempo que se actualiza la ventana, el usuario recibe una notificación avisándole de que se han encontrado nuevos recursos, de esta manera aún cuando no este usando la aplicación.

Si el usuario desea acceder a la información sobre un recurso en concreto puede hacerlo pulsando sobre el recurso del cual desea obtener más información. Al hacer esto, la aplicación abrirá una nueva ventana en la que puede consultar la información que exista en la base de datos sobre el recurso o área seleccionada. Esta ventana está formada por dos áreas, en la parte superior hay un carrusel de imágenes con las imágenes descriptivas del recurso o áreas seleccionada, en caso de que no tuviera ninguna se mostraría la misma imagen que se usa para el listado de la ventana principal. Después del carrusel se muestra al usuario toda la información asociada al recurso o al área, el usuario podrá desplazar el texto si la cantidad de información es demasiado grande para el espacio del que se dispone.

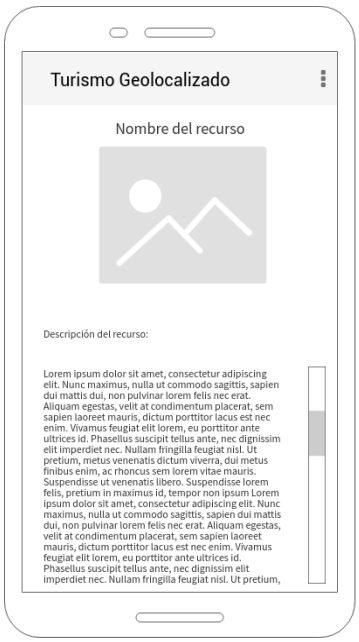

Ilustración 16: Detalle de un recurso geolocalizado

Ilustración 17: Mapa con la ubicación de los recursos

Como se ha comentado los usuario también pueden consultar la localización de los recursos geográficos sobre un mapa. Al pulsar sobre el botón correspondiente se abre una nueva ventana en la que aparecerán todos los recursos que tenemos en la base de datos. Cada recurso aparece como un marcador de posición que al pulsar sobre el muestra el nombre del recurso así como la distancia que lo separa de la posición del usuario.

Por último, desde cualquier ventana se puede acceder a la pantalla de configuración de la aplicación usando el menú situado en la esquina superior derecha de la pantalla. En esta ventana se pueden modificar varios parámetros como el nombre con el que nos hemos conectado a la aplicación *Owntracks* o si queremos mostrar o no notificaciones.

En esta ventana el usuario dispondrá de un campo de texto donde podrá modificar su nombre de usuario y un botón para activar y desactivar las notificaciones que muestra la aplicación ya que estas puede que resulten molestas para algunos usuarios.

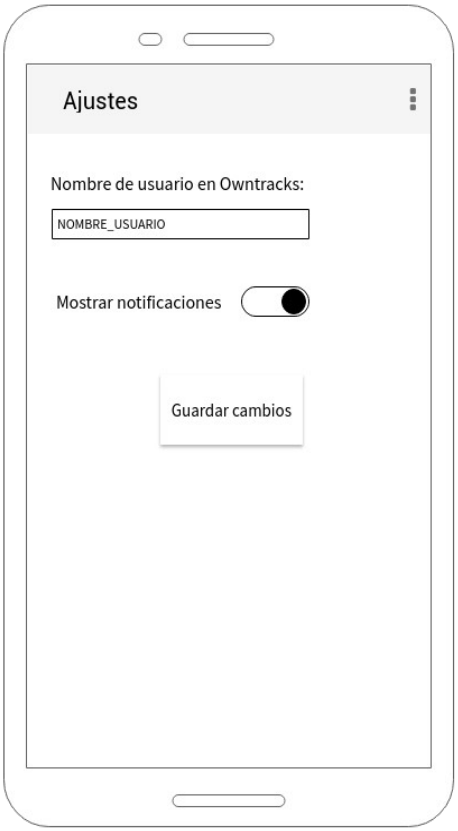

Ilustración 18: Ventana con los ajustes de la aplicación

5.2 Especificación de los mensajes

A la hora de transmitir esta información entre los distintos componentes de nuestra solución debemos definir los mensajes que se emplearán en este cometido. Estos serán mensajes de texto pero los datos que contengan deberán tener un formato concreto.

Como ya se ha comentado, tenemos cuatro mensajes diferentes aunque todos contienen datos similares por lo que los tipos de datos serán compartidos por todos los mensajes. Así, el nombre de los recursos o de las áreas, la descripción, la URL de la fotografía y el nombre de los usuarios serán conjuntos de caracteres alfanuméricos independientemente de en que mensaje se utilicen.

Por otro lado las coordenadas geográficas, serán números reales que deberán seguir el siguiente formato:

- latitud: número real con valores entre -90 y 90 para indicar latitud norte y sur respectivamente.
- longitud: número real con valores entre -180 y 180 para indicar longitud oeste y este respectivamente

Por último, cuando un usuario entra o sale de un área se genera otro tipo de mensaje, para indicar si el usuario ha entrado o salido del área geolocalizada se usarán los valores “E” y “S” para la entrada y la salida respectivamente.

5.2.1 Nuevo usuario y actualización de las áreas geolocalizadas

Cuando un nuevo usuario entra por primera vez en la aplicación deberá introducir el nombre de su dispositivo, en concreto debe introducir el mismo nombre que tenga su dispositivo en la aplicación *Owntracks*.

Una vez complete este paso se genera un mensaje con el nombre que ha introducido el usuario, por lo tanto este mensaje solo contiene esta información:

- username: cadena alfanumérica que contiene el nombre del nuevo dispositivo

Cuando este mensaje llega al servidor se genera un nuevo mensaje destinado a la aplicación *O*wntracks del nuevo usuario. Este nuevo mensaje tiene la siguiente información:

- waypoints: listado de áreas geolocalizadas cada una con la siguiente información:
    - *_type*: este campo siempre debe tener el valor “waypoint” para que *Owntracks* identifique la información como un área geolocalizada
    - *desc*: cadena alfanumérica que contiene la descripción del área
    - *tst*: número entero único entre todas las áreas que sirve para identificarlas
    - *lat*: número real que indica la latitud del centro del área
    - *lon*: número real que indica la longitud del centro del área
    - *rad*: número real que indica el radio del área geolocalizada en metros
    - *id*: número entero único que sirve de identificador único de las diferentes áreas

5.2.2 Cambio de posición del usuario

Este mensaje será generado por la aplicación *Owntracks* cuando detecte cambios en la posición del dispositivo del usuario. El mensaje consta de la varias propiedades de las cuales nos interesan fundamentalmente las siguientes relativas a la posición del dispositivo:

- *lat*: número real que indica la latitud del dispositivo
- *lon*: número real que indica la longitud del dispositivo
- inregions: lista de identificadores de áreas geolocalizadas dentro de las cuales se encuentra el dispositivo, estos identificadores serán cadenas de caracteres

Owntracks también nos ofrece información sobre la altitud (alt) y velocidad (vel) del dispositivo que en este caso no son necesarias.

5.2.3 Entrada en un área geolocalizadas

Como se ha explicado en el punto 4.2 [pág. 44], cuando un usuario entre o salga de un área geolocalizada, la aplicación *Owntracks* generará un mensaje. Este contiene el nombre del área y el tipo de evento que se ha producido así como el identificador del dispositivo que ha generado el mensaje. Por lo tanto el mensaje generado por *Owntracks* tendrá las siguientes propiedades:

- *event*: propiedad que indica el tipo del evento producido. Puede tener el valor *enter* cuando se trate de un evento de entrada y *exit* en caso de que estemos ante un evento de salida.
- *tid*: cadena alfanumérica que identifica el dispositivo que ha generado el mensaje.

5.2.4 Envío de recursos y áreas geolocalizadas

El mensaje contará de un lista de recursos cada uno de los cuales puede representar un recurso geolocalizado o un área. Las propiedades que debe contener este mensaje son los siguientes:

- *nombre*: cadena alfanumérica con el nombre del recurso.
- *urlImagen:* cadena alfanumérica con la *URL* en la que se encuentra la imagen asociada al recurso.
- *info:* cadena alfanumérica con la información asociada al recurso.
- *puntoGeografico*: este objeto contiene la longitud y la latitud del recurso o el área:
  - *latitud*: número real que indica la latitud del dispositivo.
  - *longitud*: número real que indica la longitud del dispositivo.
- *ImagenesDescriptivas:* listado de las *URLs* de las imágenes descriptivas del recurso o área.

5.3 Patrón Publicador – Subscriptor

A la hora de transmitir estos mensajes se ha decidido emplear una patrón de comunicaciones de tipo publicador – subscriptor.

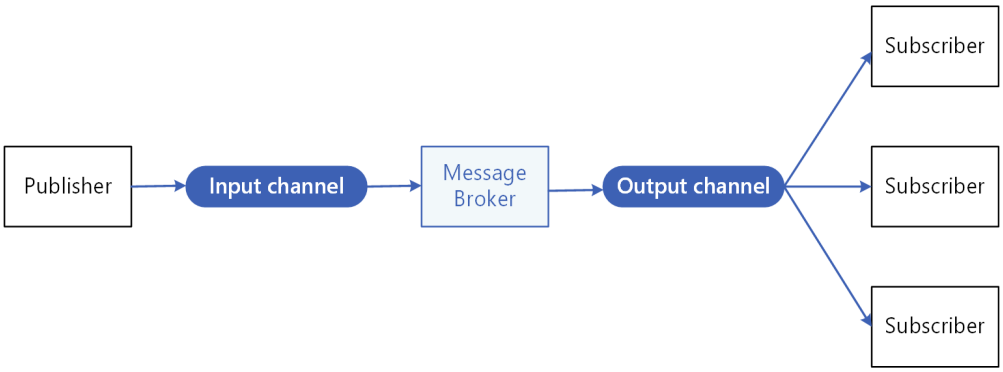

Ilustración 19: Representación del patrón publicador y suscriptor. Fuente: Microsoft

Como vemos en la ilustración 18, en los sistemas que emplean este patrón de mensajería, los dispositivos que intervienen en la comunicación pueden adoptar dos roles bien diferenciados (dejemos a un lado al *broker* por el momento), los dispositivos pueden adoptar los roles de publicador y subscriptor pudiendo tener al mismo tiempo los dos.

Un publicador es cualquier elemento que participe en la comunicación enviando mensajes mientras que los suscriptores son aquello que los reciben. La diferencia con otros patrones de comunicación es que en este la comunicación no es directa, los publicadores no envían los mensajes a un destinatario en concreto sino que los publican para todos aquello que estén suscritos a un tema. En el otro lado están los suscriptores que para poder recibir mensajes deben suscribirse a un tema o *topic* del cual quieran estar informados.

Además existe un tercer elemento cuyo papel es hacer de intermediario entre los publicadores y los suscriptores. Este mediador o *broker* registrará los diferentes temas que se creen y en los que se publicarán los diferentes mensajes y se encargará de gestionarlos para que cada suscriptor reciba los mensajes de los temas a los que se ha suscrito.

En el patrón publicador-subscriptor, un cliente que publica un mensaje está desacoplado de los otros clientes que lo reciben, es decir, los clientes no conocen de la existencia del resto de clientes (Hillar, G. C. 2017)

Esta aproximación a las comunicaciones nos permite crear un sistema muy desacoplado ya que no hay necesidad de que los productores y los suscriptores tengan información los unos de los otros por lo que se pueden producir cambios en un publicador sin que esto afecte a los suscriptores, esto es se produce un desacoplamiento entre los sistemas que participan de en la comunicación (Microsoft, 2018). De esta manera mejoramos la mantenibilidad, así como la escalabilidad del sistema.

Al tratarse de un sistema con un gran grado de desacoplamiento, resulta mucho más sencillo ampliarlo o modificarlo según las necesidades que tengamos ya que, los componentes que participan en la comunicación no necesitan saber nada los uno de los otros por lo que podríamos realizar cambios en ellos sin que el resto de la red se viera afectada. Por ejemplo, si quisiéramos cambiar el servidor o añadir otro servidor al sistema, para los cliente este cambio sería transparente y pasaría completamente desapercibido. De manera análoga, un cambio en los clientes sería imperceptible para el servidor.

5.3.1 Limitaciones

De lo visto anteriormente, podemos observar varias limitaciones de la arquitectura publicador – subscriptor. La primera y más evidente de ellas es que esta arquitectura de comunicaciones no permite la comunicación de forma bidireccional entre el servidor y los clientes, esto es una consecuencia directa de su principal ventaja, el desacoplamiento entre publicador y subscriptor.

Esto puede suponer un problema en el caso de que un subscriptor deba comunicar su estado al servidor, o enviar un acuse de recibo al mismo. Para solucionar este problema tanto el servidor como los clientes deberán ser publicadores y suscriptores al mismo tiempo

Otro de los problemas que presenta este tipo de arquitecturas de comunicaciones es que el orden en que los suscriptores reciben los mensajes no está asegurado y puede no ser el mismo en que fueron enviados. A la hora de diseñar un sistema con este tipo de arquitectura es fundamental asegurarse de que se elimina cualquier dependencia en el orden en que se tratan los mensajes (Microsoft, 2018).

5.4 Patrones de diseño

Como veremos más adelante, tanto el servidor web como la aplicación móvil están formados por múltiples componentes que interactúan entre si para conseguir una funcionalidad determinadas. El objetivo es conseguir una solución modular que nos permita realizar modificaciones en la aplicación sin que sea necesario un gran esfuerzo en tiempo o recursos.

Para ellos, se han utilizado varios patrones de diseño software que nos proporcionan soluciones ya probadas para problemas específicos. Esto no solo ahorra tiempo de desarrollo si no que incrementa la calidad del código desarrollado al ser técnicas más que probadas y que hacen que nuestro programa sea mucho más fácil de entender por otros desarrolladores lo que hace que sea mucho más sencillo de mantener.

En nuestro caso los patrones que se han empleado son los siguientes:

5.4.1 Patrón fachada

Este patrón de diseño entra dentro de la categoría de patrones estructurales, y su aplicación en el diseño de la solución nos permite proporcionar una interfaz simplificadas para acceder a funcionalidades complejas. El patrón fachada se emplea para ocultar la complejidad de un sistema y hacer que sea más fácil de implementar

En nuestro servidor tendremos componentes (clases) que nos ofrecerán funcionalidades para acceder a la base de datos o para realizar operaciones con coordenadas geográficas. Estas clases nos permiten realizar ciertas operaciones sin tener que preocuparnos por como se llevan a cabo estas operaciones.

Por ejemplo, si queremos obtener una serie de recursos de la base de datos, llamaremos al método *ObtenerRecursosGeolocalizados* de la clase *PersistenciaDAOImpl.* Internamente este método realizará las consultas necesarias , controlará los posibles errores y nos devolverá una serie de datos con los que podremos trabajar pero la implementación de esta funcionalidad pasa desapercibida para cualquiera que utilice este método de manera que podemos cambiarla sin tener que modificar aquellos componentes que la utilicen.

Desde fuera, la funcionalidad para obtener recursos de la base de datos se percibe como una única operación aunque internamente no lo sea. Esto nos permite desentendernos de su implementación y de las dependencias que esta

tenga ya que será en la clase *PersistenciaDAOImpl ,* y solo en ella, donde tengamos en cuenta todos estos detalles, de este modo sabemos que si queremos realizar cambios en la forma que accedemos a la base de datos solo hay una clase que debemos modificar. Es decir, gracias al patrón fachada conseguimos reducir el acoplamiento entre los distintos componentes de la solución haciendo que las modificaciones sean mucho más sencillas de llevar a cabo.

Esto se consigue mediante el empleo de interfaces en las que definiremos que métodos deberá tener cualquier clase que implemente la interfaz, creando de este modo un contrato entre las clases que implementan la interfaz y el resto de componentes de la aplicación. Volviendo al ejemplo anterior, la clase *PersistenciaDAOImpl* deberá implementar la interfaz *PersistenciaDAO* donde habremos definido los métodos que deben ser implementados, en un momento dado puede surgir la necesidad de hacer otra implementación de estos métodos por ejemplo porque vamos a trabajar con varias bases de datos diferentes. En este caso se crearía otra clase que implementara la interfaz *PersistenciaDAO* solo que esta tendría una implementación diferente de los métodos de la interfaz.

De esta manera, un componente que use *PersistenciaDAOImpl,* puede cambia a la nueva implementación sin ningún tipo de problemas ya que ambas tienen los mismos métodos, o dicho de otra manera, cumplen con el mismo contrato (interfaz) aunque después la implementen de formas diferentes. Es decir, conseguimos ocultar la implementación ofreciendo siempre la misma interfaz al resto de componentes.

5.4.2 Patrón *singleton*

Otro de los patrones utilizados es el patrón *Singleton*. Este patrón se utiliza para asegurar que una clase tendrá una única instancia en cualquier momento mientras se proporciona un punto de acceso global a la instancia de modo que podemos acceder a esta desde cualquier punto de nuestro programa.

En nuestra solución se ha aplicado este patrón para el componente *ProveedorPropiedades,* cómo veremos en el siguiente punto, este componente se encarga de extraer diferentes valores de configuración necesarios para la aplicación de un fichero de texto. Gracias a la aplicación de este patrón, podemos emplear este componente desde cualquier otro de la aplicación inicializándolo una única vez.

5.4.3 Patrón Modelo-Vista-Controlador

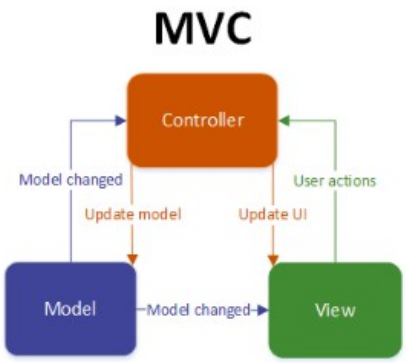

Ilustración 20: Relación entre los componentes del patrón vista-modelo-controlador. Fuente: https://medium.com/

En la aplicación desarrollada en Android se utiliza el patrón arquitectónico Modelo-Vista-Controlador, este patrón utiliza tres componentes:

- El modelo: Representa los datos que maneja nuestra aplicación, en este caso los recursos y las áreas geolocalizadas junto con todos sus datos.
- La vista: Es la interfaz gráfica, la encargada de mostrar al usuario el estado del modelo. En una aplicación Android, el diseño de la interfaz se define en archivos *XML*.
- El controlador: Se encarga de controlar la vista respondiendo a los eventos que se generen en la vista modificando el modelo según las acciones del usuario en la interfaz. En nuestro caso, al ser una aplicación Android, los controladores serán clases Java que extenderán de la clase *Activity* estas clases se conocen como *activities* o actividades.

Este patrón nos permite separar la lógica de negocio de la aplicación de la lógica de la vista, de esta manera a la hora de realizar modificaciones estas se acotan solo a uno de los componentes sin que el resto se vean afectados. Por ejemplo, si cambiamos el diseño de una ventana de la interfaz de usuario puede que debamos hacer algunas modificaciones en el controlador pero los cambios no deberían afectar a ningún otro componente. De esta manera conseguimos que la aplicación sea de mayor calidad y mucho más mantenible.

6. Implementación del Servidor

En este apartado se van a comentar los detalles de la implementación del servidor, veremos como se han desarrollado las distintas funcionalidades que componen esta parte de la solución y como se produce la interacción entre todas ellas.

El servidor se ha implementado usando el lenguaje de programación *JAVA* 1.8 y usando una base de datos relacional *SQLite* por lo que en este apartado veremos los detalles de las clases *JAVA* que componen la aplicación del servidor y se explicará como interactúa este tanto con la base de datos como con las aplicaciones móviles.

6.1 Modelo de datos

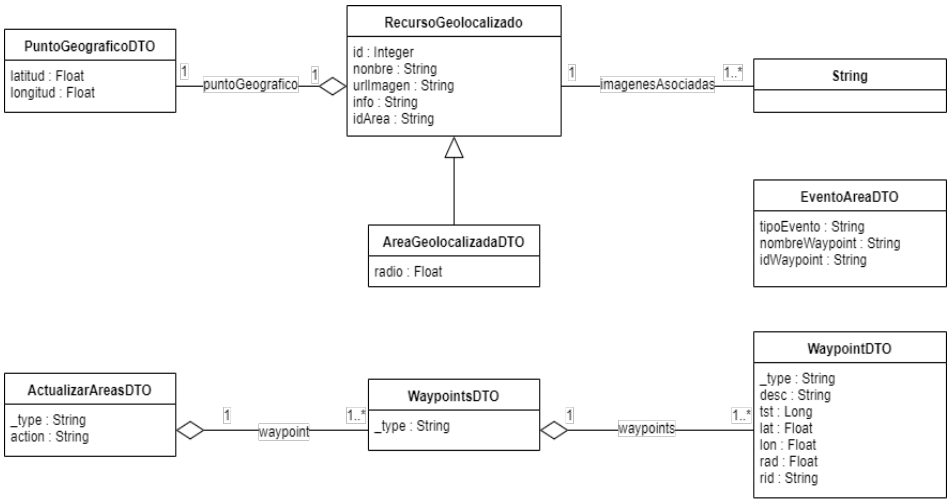

Ilustración 21: Modelo de datos. Fuente: Elaboración propia.

Para trabajar con los datos descritos en el punto 4.1, se han creado las siguientes clases :

- *PuntoGraficoDTO:* Este objeto almacena información sobre las coordenadas geográficas de un punto usando sus valores de latitud y longitud.
- *RecursoGeolocalizadoDTO:* Objeto que contiene la toda la información sobre un recurso o un área geolocalizada, en concreto tiene las siguientes

propiedades: identificador, nombre, la *URL* de la imagen representativa del recurso, información sobre el recurso, su posición representada por un objeto de tipo *PuntoGeograficoDTO,* el identificador del área a la que pertenece el recurso y, por último, una lista de *URLs* de imágenes relacionadas con el recurso.

- *AreaGeolocalizadaDTO:* Este objeto es una especialización de *PuntoGeograficoDTO* que, además de todas las propiedades que tiene un recurso, también tiene de una propiedad que indica el radio del área geolocalizada.

- *EventoAreaDTO:* Utilizamos este DTO para gestionar la información que llega al servidor cuando un usuario entra o sale de un área geolocalizada, las propiedades que tiene este objeto son: el tipo de evento producido, el nombre del área y su identificador.

- *WaypointDTO:* Representa un área geolocalizada tal como están definidas en la aplicación *Owntracks,* esto es así porque usaremos este tipo de objeto para poder informar a la aplicación de las áreas geolocalizadas que tenemos, por ello las propiedades de este objeto deben ser las mismas que se definen en la documentación de *Owntracks* para que una vez le mandemos la información sea capaz de extraer los datos y crear las áreas en la aplicación. Las propiedades de este objeto son las siguientes:

    - *_type:* Para identificar a este objeto como un área este campo debe contener siempre el valor *waypoint.*

    - *desc:* Descripción del área, opcionalmente este campo se puede dejar sin valor.

    - *lat:* Latitud del punto que representa el centro del área

    - *lon:* Longitud del punto que representa el centro del área.

    - *rad:* Tamaño del radio del área en metros.

    - *tst:* Valor numérico que representa la fecha y hora en la que fue creada el área.

    - *rid:* Identificador de la región.

- *ActualizarAreaDTO:* Para poder actualizar las áreas geolocalizadas en la aplicación *Owntracks* con la información que tenemos en la base de datos usamos este tipo de objeto, en la propiedad *_type* debemos especificar el valor *cmd* y la propiedad *action* debe tener el valor *setWaypoints*. Por último

tiene un objeto de tipo *Wypoints* que contiene una lista de objetos *WaypointDTO*. De esta manera cuando esta información llegue a *Owntracks* sabrá que la información que contiene este objeto representa una serie de áreas que debe crear.

6.2 Implementación de la aplicación del servidor

El primer componente que debemos abordar es el punto de entrada al programa, esto es el punto donde se comienzan a ejecutar las primeras instrucciones al producirse el arranque del sistema.

En nuestro caso, este punto de entrada se encuentra en la clase *InicioServidor*. Esta clase contiene el método *main* el cual es el primer método en ejecutarse en un programa JAVA.

Una vez se lanza la aplicación, este método inicializa una instancia de la clase *ProveedorPropiedades* para poder obtener los valores de configuración necesarios para poder iniciar el servidor. A continuación se obtienen estos valores y se inicia el proceso de arranque del servidor, para ello, inicializamos el acceso a la base de datos creando un objeto de tipo *PersistenciaDAO* usando la ruta que se ha proporcionado al iniciar la aplicación. Este nuevo objeto establecerá una conexión y quedará a la espera de recibir peticiones para acceder a la base de datos.

Una vez tenemos el servicio de persistencia y se ha establecido la conexión con la base de datos, podemos crear un objeto de la clase *ServicioMQTT*. Este nuevo servicio se encargará de recibir y enviar mensajes a la aplicación móvil a través del broker MQTT. Para inicializar este servicio debemos proporcionarle una referencia al objeto de tipo *PersistenciaDAO* creado anteriormente

Tras haber creado este objeto de tipo *ServicioMQTT* el servidor está iniciado y a la espera de recibir mensajes. En caso de que se produzca algún tipo de error durante la puesta en marcha, el usuario será informado mediante mensajes en la consola.

6.3 Proveedor de Propiedades

Como hemos visto, la clase *ProveedorPropiedades* nos proporciona la funcionalidad para poder obtener valores desde un archivo de texto, este archivo se ha llamado *config.properties* y consta de una serie de pares clave-valor como los siguientes:

```
#Archivo de configuracion del servidor
ruta_base_datos=/home/db/info.db
distancia_max_busqueda=50

#Configuracion de la conexion con el broker MQTT
broker_url=tcp://m24.cloudmqtt.com:13846
broker_usuario=user
broker_contrasena=password
```

Cada una de estas propiedades se pueden consultar mediante el proveedor de propiedades llamando al método *obtenerPropiedad,* al cual debemos pasar el nombre de la propiedad que queremos obtener y nos devuelve su valor en el archivo de configuración o *null* si la propiedad no se encuentra.

La clase *ProveedorPropiedades* implementa el patrón *singleton* lo que implica que solo va a existir una instancia de la clase en todo momento y este objeto podrá ser accedido desde cualquier otro componente de la aplicación. Esto nos permite realizar la configuración necesaria del archivo de propiedades una única vez con la consiguiente mejora de rendimiento.

6.4 Persistencia *DAO*

El *DAO (Data Access Object /* Objeto de Acceso a Datos*)* de persistencia es el responsable de establecer conexiones con la base de datos y de gestionar las peticiones que se deben realizar a la misma para obtener los recursos y áreas geolocalizadas, también debe ser capaz de gestionar correctamente los posibles fallos que se produzcan al utilizar la base de datos de manera que la aplicación falle de la mejor manera posible entendiendo esto como que, en caso de fallo, la aplicación debe quedar en un estado en el cual pueda seguir funcionando y donde los datos no se modifican de forma errónea. Por ello es importante gestionar las excepciones que se pueden producir empleando bloques de código *try* y *catch.*

El constructor de esta clase, toma como parámetro la ruta de nuestro sistema en la que se encuentra el el fichero que contiene la base de datos como vimos al describir el proceso de arranque del servidor e intenta realizar una conexión con la base de datos para comprobar que esta se encuentra disponible, que la ruta es la correcta y que es posible conectarnos a ella.

Para llevar a cabo la conexión se utiliza el método *ConectarBD*. Este método emplea la funcionalidad que nos proporciona la librería *SQLite* para acceder a la

base de datos, en caso de que se produzca un erro la excepción es tratada en el bloque *catch* registrando el error y previniendo que el servidor deje de funcionar.

En el siguiente fragmento de código se observa la estructura para la detección y el tratamiento de los errores:

```
//creamos la conexión con la base de datos
conn = null;
try {
      conn = DriverManager.getConnection("jdbc:sqlite:" + ruta);
} catch (Exception e) {
      e.printStackTrace();
      logger.error(e.getMessage());
}
```

Si la conexión con la base de datos se realiza correctamente, también se registra esta información para poder tener una traza del funcionamiento del servidor.

Por otro lado, el servicio de persistencia nos ofrece métodos para acceder a los datos almacenados en la base de datos, estos métodos son los siguientes:

obtenerAreaGeolocalizada: Este método nos permite obtener un área geolocalizada a partir de su nombre, realizando una consulta sobre la tabla *areas_geolocalizadas* y creando un objeto de tipo *AreaGeolocalizada* con el resultado de la consulta a la base de datos.

ObtenerAreasGeolocalizadas: A diferencia del método anterior, este obtiene toda la información sobre las áreas geolocalizadas sin aplicar ningún tipo de condición, una vez se han obtenido todos los datos, se crea una lista con los objetos de tipo *AreaGeolocalizada*.

ObtenerRecursosGeolocalizados: Como su nombre indica este método se encarga de obtener la información sobre los recursos geolocalizados que tenemos en la base de datos, el método recibe una lista con los nombre de las áreas que contienen los recursos que queremos obtener. De esta manera podemos obtener todos los recursos que se encuentren dentro de un área determinada o por el contrario aquellos que no estén asociados a ninguna si la lista no contiene ningún valor. El método realiza una consulta *SQL* sobre la tabla *recursos_geolocalizados* que devuelve un conjunto de filas cada una de las cuales contendrá la información de un recurso concreto.

Para realizar la consulta sobre la base de datos se utiliza el método *executeQuery* que nos devuelve los resultados en una estructura de datos de tipo *ResultSet*. Esta estructura almacena las filas obtenidas por la consulta *SQL* pero

resulta poco práctica de utilizar y modificar. Para solucionar este problema, una vez hemos obtenido los resultados de la consulta recorremos el *ResultSet* y, para cada fila creamos un objeto de tipo *RecursoGeolocalizado.* Posteriormente, rellenamos este recurso con su nombre, la URL de la imagen, el texto que contiene la descripción del recurso y un identificador único que nos ayuda a identificar este recurso.

Además, como hemos visto en el apartado del modelo de datos, cada objeto de tipo *RecursoGeolocalizado* tiene a su vez un objeto de tipo *PuntoGeográfico* que almacena la información sobre su posición geográfica por lo que se crea un objeto de este tipo para cada resultado obtenido por la consulta, en este objeto se especifican las propiedades latitud y longitud del recurso geolocalizado. Asimismo, tanto los recursos como las áreas pueden llevar asociadas imágenes descriptivas que se almacenan en la tabla *imagenes*. Por lo tanto, al recuperar un recurso o un área desde la base de datos también obtenemos las *URLs* de estas imágenes como un listado que asignaremos a la propiedad *iamgenesDescriptiva* del objeto *RecursoGeolocalizado.*

6.5 Servicio *JSON*

A la hora de transmitir mensajes ente el servidor y la aplicación móvil, debemos codificar los objetos JAVA en un formato como *JSON* que nos permite su transmisión empleando el protocolo *MQTT*, a esta operación de codificación se la conoce como serialización.

La clase *ServicioJSON* nos proporciona métodos para serializar y deserializar nuestros objetos del modelo de datos de forma que podamos tanto recibir como enviar información a la aplicación móvil. Para llevar a cabo estas operaciones vamos a emplear la librería *Gson* que nos proporciona funcionalidades necesarias para serializar y deserializar objetos JAVA.

Las operaciones que nos proporciona este componte del servidor son las siguientes:

- *serializarRecursos*: Transforma un objeto en un texto en formato *JSON*.
- *deserializarEventoArea*: Transforma una cadena de caracteres en formato *JSON* en un objeto de tipo *EventoArea*.
- *serializarAreaGeolocalizada*: Transforma un objeto de tipo *AreaGeolocalizada* en una cadena de caracteres en formato *JSON*.

- *obtenerJsonActualizarAreas:* Transforma una lista de objetos de tipo *AreaGeolocalizada* en un mensaje en formato *JSON* requerido por la aplicación *Owntracks* para poder actualizar las áreas geolocalizadas (*waypoints*) de la aplicación.
- *deserializarPuntoGeografico*: Transforma un texto en formato *JSON* en un objeto de tipo *PuntoGeografico*.
- *deserializarMensajeGeolocalizacion*: Transforma un texto en formato JSON en un objeto de tipo *MensajeLocalizacion*.

A la hora de implementar este servicio se ha empleado la librería *Gson* a la hora de serializar objetos *JAVA* lo cual nos permite no tener que implementar la conversión a *JSON* para cada uno de los tipos de objetos con los que vamos a trabajar. Esta librería nos proporciona el método *toJson* el cual recibe como parámetro un objeto y lo transforma en una cadena de caracteres que representa a ese objeto en formato *JSON,* de esta manera usaremos el método *tojson* en los métodos del servicio dedicados a serializar objetos para poder enviar el mensaje *JOSN* resultante a la aplicación de geolocalización o a la aplicación *Owntracks,* estos métodos son: *serializarRecursos, serializarAreaGeolocalizada y obtenerJsonActualizarAreas.* La utilización de la librería *Gson* en estos métodos resulta trivial ya que únicamente debemos crear un objeto de tipo *Gson* y llamar al método *toJson* descrito anteriormente como podemos ver a continuación:

```
public String serializarRecurso(Object recursos) {
      Gson gson = new Gson();
      String resJson = gson.toJson(recursos);
      return resJson;
}
```

En el caso del método *obtenerJsonActualizarAreas,* antes de llamar al método *toJson* hay que convertir el listado de áreas que recibe el método en un objeto de tipo *ActualizarAreas* que es el que finalmente se convierte en un texto *JSON,* para ello se crea un objeto de este último tipo y se rellenan sus datos de la siguiente manera:

```
ActualizarAreas actualizarAreas = new ActualizarAreas();
            for(AreaGeolocalizada area : areas) {
                  Waypoint w = new Waypoint();
                  w.setDesc(area.getNombre());

      w.setLat(area.getPuntoGeografico().getLatitud());

      w.setLon(area.getPuntoGeografico().getLongitud());
                  w.setRad(area.getRadio());
```

```
                w.setRid(String.valueOf(area.getId()));
                w.setTst(area.getId());

        actualizarAreas.getWaypoints().getWaypoints().add(w);
            }
```

Por otro lado, el proceso inverso al descrito hasta ahora resulta un poco más complejo ya que aunque para transformar un texto *JSON* en un objeto *Java* también se utiliza la librería *gson* pero el proceso no resulta tan inmediato como al hacer la operación contraria.

En primer lugar se debe crear un objeto de tipo *JsonParse*, este objeto forma parte de la librería *gson* y nos permite leer el texto en formato *JSON* y generar un nuevo objeto de tipo *JsonObject* el cual también forma parte de la citada librería. A partir de este objeto podremos extraer todas las propiedades que necesitemos para crear nuestros objetos.

Este proceso se sigue en los métodos *deserializarEventoArea*, *deserializarNuevoUsuario, deserializarMensajeGeolocalizacion y deserializarPuntoGeografico,* a continuación se muestra este último método a modo de ejemplo:

```
public PuntoGeografico deserializarPuntoGeografico(String json){
      JsonParser parser = new JsonParser();
      JsonObject jsonObject = null;
      try {
      jsonObject = parser.parse(jsonString).getAsJsonObject();
      } catch (JsonSyntaxException e) {
            log.error(e);
      }

      float latitud = jsonObject.get("lat").getAsFloat();
      float longitud = jsonObject.get("lon").getAsFloat();
      return new PuntoGeografico(latitud, longitud);
}
```

Cómo podemos ver en el fragmento anterior, después de obtener un objeto de tipo *JsonObject* a partir del texto *JSON* extraemos de el las propiedades latitud y longitud para lo cual tenemos que usar las claves “lat” y “long” respectivamente. Finalmente podemos crear un objeto *PuntoGeografico.* Este proceso es seguido de igual forma por el resto de métodos comentados anteriormente.

6.6 Servicio de Operaciones Geográficas

Este servicio proporciona métodos para realizar operaciones con coordenadas geográficas, en concreto nos ofrece la funcionalidad necesaria para calcular la distancia entre dos puntos geográficos y para obtener todos los puntos geográficos a cierta distancia de otro punto dado.

El método para calcular la distancia entre dos coordenadas se llama *obtenerDistancia,* toma como parámetros dos puntos geográficos y devuelve la distancia aproximada en metros entre ambos.

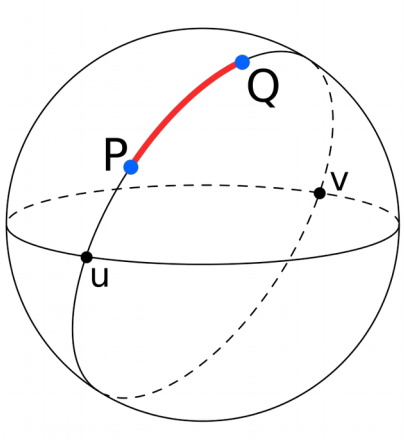

Ilustración 22: Representación de la distancia ortodrómica. Fuente: Wikipedia

Para calcular la distancia debemos calcular la llamada distancia ortodrómica, es decir la menor distancia entre dos puntos en la superficie de una esfera. Dado que la tierra no es una esfera perfecta este método introduce un error de un 0.5%, error que es perfectamente asumible para esta aplicación

La distancia se calcula como el producto del ángulo central formado por los puntos y el radio de la tierra.

Por lo tanto la distancia será $d = R \cdot \Delta\sigma$ donde $\Delta\sigma$ es el ángulo central y *R* es el radio de la tierra. Existen varios métodos para calcular el ángulo central entre dos puntos en una esfera, en nuestro caso vamos a emplear la fórmula del semiverseno, también conocida como *haversine formula* en inglés:

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi 1 \cdot \cos\varphi 2 \cdot \sin^2(\Delta\lambda/2)$$

donde Δφ es la diferencia de las latitudes, Δλ es la diferencia de las longitudes y donde φ1 y φ2 son la latitud del primer y segundo punto respectivamente. Aplicando la fórmula del semiverseno a al hora de obtener el ángulo central obtenemos esta ecuación:

$$c = 2 \cdot asin(\sqrt{a})$$

Tras obtener este ángulo solo resta multiplicar por el radio de la tierra para obtener la distancia. Veamos a continuación como sería este algoritmo implementado en JAVA.

El primer paso para calcular esta distancia es obtener las latitudes y longitudes en radianes de los puntos geográficos. Este paso resulta trivial ya que esta información se pasa como parámetro al método en dos objetos de tipo *PuntoGeografico.* Para conseguir estos datos en radianes empleamos el método *toRadians* que nos proporciona la clase *Math.*

```
public static double distancia(PuntoGeografico origen,
PuntoGeografico destino) {
      double lat1 = Math.toRadians(origen.getLatitud());
      double lat2 = Math.toRadians(destino.getLatitud());
      double lon1 = Math.toRadians(origen.getLongitud());
      double lon2 = Math.toRadians(destino.getLongitud());
```

Tras obtener estos datos , aplicamos la fórmula del semiverseno para obtener el ángulo central y por último la distancia entre los puntos.

```
      double latDistance = lat2 - lat1;
      double lonDistance = lon2 - lon1;

      double a = Math.pow(Math.sin(latDistance / 2), 2)
            + Math.cos(lat1) * Math.cos(lat2)
            * Math.pow(Math.sin(lonDistance / 2),2);

double c = 2 * Math.asin(Math.sqrt(a));
double distance = R * c * 1000; // Convertimos el resultado a
metros
return distance;
```

La constante “R” es el radio de la tierra medido en kilómetros, esta constante está declarada en la clase OperacionesGeometricas.

```
private static final double R = 6264.85;
```

Además de poder calcular la distancia entre dos puntos geográficos, la clase Operaciones geográficas también nos proporciona un método para obtener todos los recursos que se encuentren a menos de una cierta distancia de otro recurso dado. Este método llamado *obtenerRecusrsosEnRadio* nos devuelve un listado con objetos de tipo RecursoGeografico y tiene como parámetros el radio en metros del área de búsqueda y el punto alrededor del cual hay que realiza la misma y una lista de identificadores de las áreas dentro de las cuales deben estar los recursos. Este último parámetro es opcional y si no se proporciona se realizará una búsqueda sobre todos los recursos geolocalizados en base de datos.

En vista de la limitada cantidad de recursos con los que va a trabajar esta solución se ha empleado una aproximación por fuerza bruta, es decir, obtenemos todos los recursos geolocalizados de la base de datos y una vez los tenemos usamos el método *obtenerDistancia* que acabamos de comentar para saber a que distancia se encuentra cada recurso del punto geográfico que nos interesa. Aquellos recursos que estén a una distancia igual o menor de la deseada, se añaden a una lista de resultados que se devuelve una vez se han calculado las distancias de todos los recursos geográficos.

6.7 *Socket Factory*

Esta clase proporciona un único método para crear un objeto de tipo *SSLSocketFactory* a partir de un certificado digital, como veremos, a la hora de realizar la conexión con el broker *MQTT* se usará este objeto para que la conexión entre el servidor y el broker use *SSL* para cifrar los datos que enviemos y recibamos, de esta manera la comunicación será mucho más segura.

No vamos a entrar en los detalles de la implementación, ya que la clase SSLSocketFactory nos la proporciona *JAVA* así como la funcionalidad para obtenerla, para ello utilizaremos un certificado digital (archivo .crt), en este caso utilizaremos un certificado auto firmado, esto es, un certificado que no ha sido firmado por una autoridad certificadora. En nuestro caso esto no es necesario por lo que vamos a usar este tipo de certificados.

6.8 Servicio MQTT

Este servicio constituye el núcleo de la aplicación del servidor y usará el resto de servicios descritos anteriormente para gestionar los mensajes *MQTT* que llegan desde la aplicación móvil y desde la aplicación *Owntracks* para responder a ellos con la información necesaria.

Tal como vimos en el apartado dedicado a la clase *InicioServidor*, para inicializar el servicio que gestiona los mensajes MQTT creábamos un nuevo objeto de tipo *ServicioMQTT* pasándole como parámetros un objeto *ServicioPersistencia,* la distancia máxima de búsqueda, la URL del *broker*, el usuario con el que se tendrá que identificar el servidor en el *broker,* la contraseña y por último la ruta en la que se encuentra el certificado digital que necesitaremos para poder utilizar el protocolo *SSL* y cifrar la comunicación entre el servidor y el *broker.*

En la URL al broker deberemos especificar tanto el protocolo que vamos a utilizar para conectarnos, como ya se ha mencionado será SSL, como el puerto del broker al que deberemos conectarnos. En nuestro caso la ruta quedará de la siguiente manera: "**ssl**://tambori.dsic.upv.es:**8884**". Tanto esta información como el usuario y contraseña del gestor de comunicaciones o la ruta al certificado digital se encuentra en el fichero de configuración *config.properties* gracias a lo cual se pueden modificar en cualquier momento de forma rápida y sencilla.

Una vez invocado el constructor creamos el resto de servicios que nos van a hacer falta como son el *ServicioJSON* y el servicio O*peracionesGeográficas*. También obtenemos el certificado digital que utilizaremos a la hora de conectarnos y, por último realizamos una llamada al método *connect* para realizar la conexión con el *broker MQTT*.

6.8.1 Conexión con el broker MQTT

El método *connect* es el encargado de realizar la conexión entre el servidor y el *broker MQTT*. El primero paso es inicializar una serie de opciones de conexión, estas opciones nos sirven para establecer el tipo de conexión, en este caso se utiliza una conexión sin persistencia (*clean session*), además queremos estar siempre aunque pasemos largos periodos sin actividad, para ello usamos el método *`setKeepAliveInterval`* y le pasamos el valor cero.

Los siguientes parámetros de configuración que debemos establecer son el nombre de usuario y la contraseña, estos vendrán definidos por el broker y debemos emplearlos a la hora de realizar la conexión.

Tras esto, dentro de un bloque *try/catch,* usamos la clase *SocketFactory* para obtener un objeto de tipo *`SSLSocketFactory`* a partir del certificado que usaremos para establecer una conexión segura con el broker.

```
try{
      SSLSocketFactory factory =
      SocketFactory.getSocketFactory(crtInputStream);
      connOpt.setSocketFactory(factory);
}
```

Después, creamos un cliente MQTT la ruta a la que debe conectarse, un nombre para el cliente y por último debemos indicar un mecanismo para almacenar los mensajes que nos lleguen. En nuestro caso usaremos el mecanismo que nos proporciona la librería *Phao* por defecto.

```
clienteMQTT = new MqttAsyncClient(brokerURL, "Server",new
MemoryPersistence());
```

Para realizar la conexión tenemos que llamar al método *connect* del cliente que acabamos de crear.

```
clienteMQTT.connect(connOpt, null, new IMqttActionListener() {
    public void onSuccess(IMqttToken arg0) {
          //nos suscribimos al los temas
          subscribe(TOPIC_EVENTOS);
          subscribe(TOPIC_POSICIONES);
    }
    public void onFailure(IMqttToken arg0, Throwable arg1) {
          //en caso de fallo registrar el evento
          Logger.getLogger("ServicioMQTT").log(Level.SEVERE,
                "fallo al conectar con el servidor mqtt");
          }
    });
```

Este método intentará llevar a cabo la conexión usando las opciones que hemos configurado previamente en el objeto *connOpt.* Dependiendo del resultado del intento de conexión se ejecutará uno de estos dos métodos:

- *onSucces* se ejecutará en el caso de que se pueda establecer la conexión de forma exitosa
- *onFailure* se ejecutará si no es posible conectarse al broker MQTT por cualquier motivo.

En caso de fallo registramos el el evento, Si por el contrario nos conectamos correctamente, el siguiente paso es suscribirnos a los dos temas necesarios para recibir información sobre la localización de los usuarios y sobre las entradas y salidas de áreas geolocalizadas.

6.8.2 Suscripción a los temas

Para suscribirnos a un *topic* o tema la clase *ServicioMQTT* proporciona el método *suscribirse* al cual podemos pasar una cadena de caracteres, que deberá ser el nombre del tema al cual nos queremos conectar.

En la implementación de este método, usaremos el objeto *clienteMQTT* para llevar a cabo la suscripción al tema seleccionado pasándole como parámetros el nombre del tema y la calidad que queremos para la suscripción. Esta calidad define la garantía de entrega que tendrán los mensajes, puede tener tres valores:

- 0: No hay garantía de entrega, el receptor no hace acuso de recibo y el mensaje no se almacena en el emisor ni es reenviado por este.
- 1: Garantiza que el mensaje será entregado al menos una vez, el emisor del mensaje lo guarda hasta que el receptor confirma su recepción.
- 2: Similar al caso anterior con la diferencia de que este nivel de calidad garantiza que el mensaje llegará al destinatario una única vez.

6.8.3 Recepción de mensajes

A partir del momento en que el servidor se suscribe a un tema, comenzamos a recibir los mensajes que se publiquen en el mismo. Para poder recibir estos mensajes, nuestra clase debe implementar el método *messageArrived.* Este método recibe dos parámetros, el primero es una cadena de caracteres con el nombre del tema en el que se ha publicado el mensaje, el segundo parámetro es un objeto del tipo *MqttMessage* el cual tiene el contenido del mensaje MQTT.

```
public void messageArrived(String topic, MqttMessage
mqttMessage){
        if(topic.equals(NUEVO_USUARIO_TOPIC)) {
           tratarMensajeNuevoUsuario(mqttMessage);
        }else {
            String nombreUsuario = topic.split("/")[2];
            if(topic.matches("owntracks/.*/.*/event")) {
                tratarMensajeZona(nombreUsuario, mqttMessage);
            }else {

    tratarMensajeLocalizacion(nombreUsuario,mqttMessage);
            }
        }
```

Recordemos que los temas a los que nos hemos suscrito son:

- “owntracks/NOMBRE_USUARIO/NOMBRE_DISPOSITIVO” para recibir información cada vez que un usuario cambie de posición
- “owntracks/NOMBRE_USUARIO/NOMBRE_DISPOSITIVO/event” para recibir información cada vez que un usuario entre o salga de un área geolocalizada.

- “owntracks/newuser” para recibir información cada vez que un nuevo usuario comience a utilizar la aplicación móvil.

En primer lugar tras recibir un nuevo mensaje comprobamos si el tema en el que se ha publicado es el dedicado a recibir información sobre los nuevos usuario, de ser así se llamará al método *tratarMensajeNuevoUsuario* en caso de que el tema sea distinto comprobamos si se trata del tema en el que se publican los mensajes sobre los cambios de localización de los usuarios o si es el *topic* en el que se publican los mensajes cada vez que un usuario entra o sale de un área geolocalizada. En el primero de los casos se llamará al método *tratarMensajeLocalizacion* mientras que en el segundo caso se llamará al método *tratarMensajeZona.* Ambos métodos reciben como parámetros el nombre del usuario que ha generado el mensaje obtenido a partir del *topic* así como el cuerpo de dicho mensaje.

Veamos ahora estos métodos dedicados a tratar los diferentes tipos de mensajes que puede recibir el servidor.

Como ya hemos dicho el método *tratarMensajeNuevoUsuario* se encargará de procesar los mensajes generados por la aplicación móvil cuando un usuario la utiliza por primera vez e ingresa su nombre de usuario. El objetivo de este método es notificar a la aplicación *Owntracks* que esté empleando el nuevo usuario de las áreas geolocalizadas que están definidas en el sistema, para ello primero obtenemos del mensaje *JSON* recibido el nombre del nuevo usuario utilizando el método *deserializarNuevoUsuario* de la clase *ServicioJSON.* Después obtenemos una lista de *AreaGeolocalizada* a través del método *obtenerAreasGeolocalizadas* del servicio de persistencia. Finalmente se genera y se publica el mensaje *JSON* que va a ser publicado, este mensaje contendrá la información sobre las áreas geolocalizadas que hemos almacenado en el listado en un formato entendible por la aplicación *Owntracks* de manera que cuando reciba el mensaje será capaz de crear estas mismas áreas en la aplicación. El tema en el que se publicará el mensaje tendrá el siguiente formato: “owntracks/xjtrriwb/NOMBRE_NUEVO_USUARIO/cmd”

A continuación se muestra la implementación descrita:

```
String msg = new String(mqttMessage.getPayload());
```

```
String nuevoUsuario =
servicioJSON.deserializarNuevoUsuario(msg));

List<AreaGeolocalizada> areas =
servicioPersistencia.obtenerAreasGeolocalizadas();

String json = servicioJSON.obtenerJsonActualizarAreas(areas);

MqttMessage message = new MqttMessage();
message.setPayload(json.getBytes());
Publish("owntracks/xjtrriwb/"+nuevoUsuario+"/cmd", message)
```

Para procesar los mensajes relativos los cambios de posición de los usuarios se ha creado el método *tratarMensajeLocalizacion,* este método se encargará de publicar nuevos mensajes con los recursos y las áreas geolocalizadas que estén próximos al usuario según su posición. Para conseguir esto lo primero que hace este método es obtener un objeto de tipo *MensajeGeolocalizacion* a partir del mensaje *JSON* recibido utilizando el método *deserializarMensajeGeolocalizacion*. En este objeto tendremos la información tanto de las áreas geolocalizadas dentro de las cuales se encuentra el usuario como de su posición, con esta información podemos obtener un listado de objetos de tipo *RecursoGeolocalizado* gracias al método *obtenerRecursosEnRadio* del servicio *operacionesGeograficas.* Tras esto solo queda formar el nuevo mensaje y publicarlo en el topic "owntracks/NOMBRE_DEL_USUARIO/resources".

A continuación se muestra la implementación descrita:

```
String mensaje = new String(mqttMessage.getPayload());

MensajeLocalizacion msgLoc =
servicioJSON.deserializarMensajeGeolocalizacion(mensaje);

if(msgLoc.getPuntoGeografico() != null) {

List<RecursoGeolocalizado> recursos =
operacionesGeograficas.obtenerRecursosEnRadio(
      distanciaBusqueda,
      msgLoc.getPuntoGeografico(),
      msgLoc.getAreas());

MqttMessage message = new MqttMessage();

message.setPayload(servicioJSON.serializarRecurso(recursos).getB
ytes());
Publish("info/"+username+"/resources",message);
```

Por último veamos la implementación del método *tratarMensajeZona.* De manera similar al método anterior, primero transformamos el mensaje *JSON* que recibimos en un objeto de tipo *EventoArea* del cual extraeremos la información sobre el tipo de evento (entrada o salida) y nombre del área geolocalizada en la que se ha producido el evento. Tras esto obtenemos la información sobre el área geolocalizada empleando el método *obtenerAreaGeolocalizada* del servicio de persistencia, una vez hemos obtenido esta información creamos el nuevo mensaje que vamos a publicar que tendrá como contenido el objeto de tipo *AreaGeolocalizada* que acabamos de obtener para lo que se utiliza el método *serializarAreaGeolocalizada* de la clase *ServicioJSON.* Por último dependiendo del tipo de evento que estemos tratando publicaremos el mensaje en un tema determinado, si el mensaje es de tipo "enter" se publicará en el topic "info/+NOMBRE_DEL_USUARIO+/enter" mientras que si el evento es de tipo "exit" se publicará en el tema "info/+NOMBRE_DEL_USUARIO+/exit".

A continuación se muestra la implementación descrita:

```
EventoArea event = servicioJSON.deserializarEventoArea(new
String(mqttMessage.getPayload()));

String eventType = event.getEventType();
AreaGeolocalizada areaData =
servicioPersistencia.obtenerAreaGeolocalizada(
      event.getWaypointName());

MqttMessage message = newMqttMessage();
message.setPayload(
servicioJSON.serializarAreaGeolocalizada(areaData).getBytes());


if(eventType.equals("enter")) {
      Publish("info/"+username+"/enter",message);
}else {
      Publish("info/"+username+"/exit",message);
}
```

6.8.4 Envío de mensajes

A la hora de enviar mensajes desde el servidor hacía los usuarios que estén usando las aplicaciones móviles usaremos el método *publish.* En este método usaremos el objeto *clienteMQTT* para enviar un mensaje en un tema concreto.

Debemos indicar la calidad con la que queremos que se entregue este mensaje de manera análoga a como hacíamos al suscribirnos a un tema. También especificamos que no queremos que el mensaje sea retenido en el *broker*, de esta manera el *broker* no almacenará este mensaje. En caso contrario se almacenaría ese mensaje en el *topic* indicado hasta que se envié un nuevo mensaje en el mismo *topic*. Cuando un cliente se suscriba a ese tema recibirá inmediatamente el mensaje retenido.

En nuestro caso esto no resulta útil ya que la información que estamos enviando es una colección de recursos geolocalizados que se encuentra próximos a al usuario. Esta información dejará de ser válida en el momento que la posición de este cambie por lo que no tiene sentido retener este mensaje en el *broker*.

Por último usamos el objeto *clienteMQTT* para publicar el mensaje en el tema indicado. Debemos tener en cuenta que se pueden producir errores a la hora de publicar el mensaje así que debemos gestionarlos para evitar que el servidor deje de funcionar y prestar servicio.

7. Implementación de la Aplicación Móvil

La aplicación móvil se ha desarrollado para el sistema operativo Android por lo que en su desarrollo se ha empleado JAVA de igual forma que en el desarrollo del servidor. A pesar de esto, el desarrollo de aplicaciones Android tiene ciertas características propias como por ejemplo gestionar el ciclo de vida de las actividades que compondrán nuestra aplicación. ¿Pero que es una actividad en el contexto de Android? Una actividad representa una única ventana de nuestra aplicación con la que el usuario podrá interactuar para llevar a cabo cualquier acción.

Por lo tanto tendremos tantas actividades como ventanas tenga la aplicación y, cada una de estas actividades se implementa con una clase Java. Esta clase será la responsable de crear la interfaz de usuario y de responder a las acciones que el usuario realice en la ventana. El requisito indispensable para que una clase funcione como una actividad es que esta herede de la clase *Activity* que nos proporciona los métodos necesarios para poder gestionar el ciclo de vida de la actividad así como otras funcionalidades.

Teniendo en cuenta lo anterior la aplicación móvil estará compuesta por múltiples clases entre las que tendremos cuatro actividades para cada una de las pantallas de la aplicación. Como veremos más adelante estas actividades necesitan compartir datos entre ellas para lo cual se utiliza un objeto específico de *Android* llamado *Intent*.

Los objetos de tipo *Intent* nos permiten pasar información entre actividades o componentes de la aplicación además también se utilizan a la hora de iniciar una actividad, un servicio en segundo plano o un *broadcast receiver* entre otros. Por ejemplo para lanzar una nueva actividad pasandole datos usaríamos el *Intent* de la siguiente forma:

```
Intent intent = new Intent(this, Actividad.class);
startActivity(intent);
String EXTRA_MENSAJE = "com.example.app.MENSAJE";
intent.putExtra(EXTRA_MENSAJE, “Datos para la actividad”);
```

7.1 Ciclo de vida de una actividad Android

Antes de entrar en el detalle de la implementación de la aplicación móvil merece la pena detenernos a explicar el concepto "ciclo de vida" en el contexto del desarrollo de aplicaciones en Android.

Todas las actividades que componen nuestra aplicación pasan por diferentes etapas conforme los usuarios realizan acciones y navegan por la aplicación, estas etapas y a la relación que existe entre ellas se conoce como ciclo de vida.

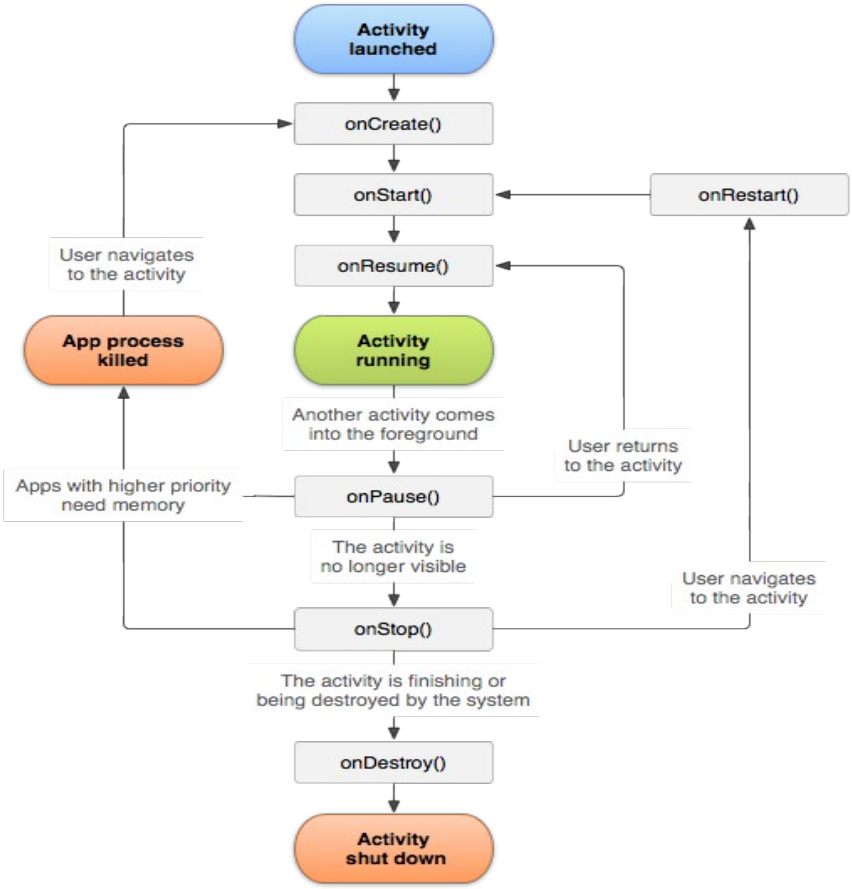

Ilustración 23: Diagrama del ciclo de vida de una actividad Android. Fuente: Android

Cuando una actividad entra en un nuevo estado, el sistema Android realiza una llamada al método correspondiente. En ese método deberemos implementar el funcionamiento que queremos que tenga la actividad al cambiar a ese estado o bien podemos dejarlo por defecto. Esto es útil para asegurarnos que la aplicación seguirá funcionando correctamente aunque se cierre de forma inesperada, por ejemplo al recibir el usuario una llamada a su teléfono mientras utiliza la aplicación.

La clase *Activity* nos proporciona los métodos necesarios para poder gestionar el ciclo de vida de nuestra aplicación, estos son: *onCreate, onStart, onResume, onPause, onStop y onDestroy.*De estos los más importantes son los encargados de gestionar la creación (*onCreate*) y la destrucción (*onDestroy*) de la actividad.

En el método *onCreate* típicamente inicializaremos los componentes necesarios para la actividad como por ejemplo, instancias de otras clases o servicios que debamos emplear en la actividad. Además, debemos especificar la interfaz de usuario que deberá mostrar la actividad, para ello emplearemos el método *setContentView.* Es también en este método donde podemos obtener datos que se le hallan pasado a la actividad a la hora de crearla. Por ejemplo si creamos una actividad para mostrar información sobre un recurso seguramente nos interese tener información sobre ese recurso en la nueva actividad como por ejemplo su identificador para poder buscarlo en la base de datos.

Por otro lado, si en algún momento el sistema decide detener la actividad bien porque el usuario ha navegado a otra ventana de la aplicación o bien porque esta se ha cerrado por algún motivo, se llamará al método *onDestroy.* En este método deberemos llevar a cabo las operaciones necesarias para que la aplicación se cierre de forma correcta. Entre estas operaciones podemos destacar: comprobar el motivo de la detención de la actividad y actuar en caso de que sea debido a un error, liberar recursos que pueda estar utilizando la actividad como pueden ser conexiones con la base de datos o guardar algún tipo de información que no queremos perder al terminar la actividad.

En nuestro caso, la gestión del ciclo de vida de las actividades es especialmente importante en dos casos: En la actividad *MainActivity* que se encarga de gestionar la ventana principal de la aplicación en la cual se muestra la información sobre los recursos geolocalizados cercanos y en la clase *ServicioSegundoPlanoMQTT* que es la encargada de notificar a la actividad *MainActivity* que se han encontrado nuevos recursos geolocalizados que debe mostrar en pantalla.

En el apartado dedicado al funcionamiento de la aplicación veremos como se ha implementado la gestión del ciclo de vida y sus eventos en estas clases y porque es necesario llevar a cabo esta implementación.

7.2 Modelo de datos de la aplicación

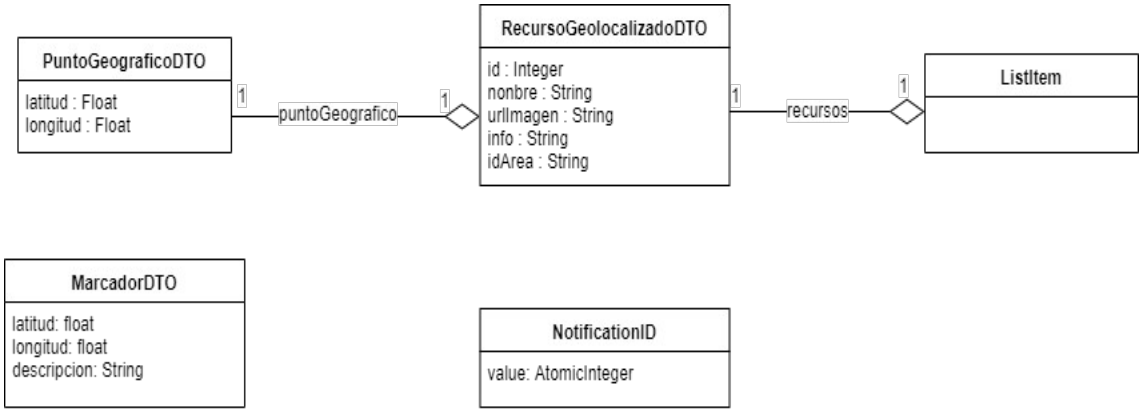

Ilustración 24: Modelo de datos de la aplicación móvil

Para la aplicación móvil se ha utilizado un modelo de datos similar al empleado en el servidor ya que, en última instancia, estamos tratando con los mismos datos: áreas y recursos geolocalizados. A diferencia de en el servidor estos dos datos los podemos tratar como un único tipo ya que la información que vamos a mostrar de los recursos y de las áreas geolocalizadas es el misma, por ello se ha creado la clase *RecursoGeolocalizadoDTO* la cual tiene las siguientes propiedades: un identificador, el nombre del recurso, la *URL* de la imagen, el texto con la información y un objeto de tipo *PuntoGeográficoDTO* .

Además esta clase contiene los métodos necesarios para poder enviar objetos de esta clase entre actividades de manera que nos sea más sencillo trabajar con los recursos. Para ello la clase *RecursoGeolocalizadoDTO* implementa la interfaz *Parcelable* por lo que deberemos implementar los métodos definidos en esta interfaz, en concreto el método *writeToParcel* que se usará para transforma un objeto de este tipo en uno de tipo *Parcel* usados para poder mover datos entre actividades.

También tenemos una clase para manejar la información que queremos mostrar en el mapa de recursos. Esta clase llamada *MarcadorDTO* almacenará para cada recurso su latitud, longitud y su descripción. De esta manera podremos situarlo sobre el mapa y seremos capaces de mostrar su descripción cuando el usuario pulse sobre el. En esta clase se ha implementado de nuevo la funcionalidad necesaria para poder pasar entre distintas actividades objetos del tipo *MarcadorDTO.*

Por último el objeto *PuntoGeográficoDTO* es análogo al visto en la implementación del servidor y contiene las propiedades latitud y longitud.

7.3 Implementación de la aplicación

En este apartado vamos a comentar los detalles de la implementación de los componentes más importantes de la aplicación, en concreto veremos como se han implementado las actividades *MainActivity, DetalleActivity* y *MapaActivity* así como el servicio de control de las notificaciones *GestorNotificaciones* y el servicio *ServicioSegundoPlanoMQTT.* También comentaremos brevemente las diferencias del resto de componentes con los del servidor.

La clase *ServicioSegundoPlanoMQTT* hereda de la clase *Service* que nos proporciona Android para poder realizar operaciones de larga duración en segundo plano, es decir, sin bloquear al resto de componentes de la aplicación. Existen en Android varios tipos de servicios que podemos utilizar, en este caso se ha optado por utilizar un servicio de tipo “iniciado” lo que nos permite que el servicio se ejecute de manera ininterrumpida en segundo plano aunque el componente que lo inicie sea destruido.

En concreto, la actividad *MainActivity* iniciará el servicio para poder recibir la información sobre los recursos cercanos al usuario. Gracias a que el servicio se sigue ejecutando en segundo plano de manera indefinida incluso aunque el usuario salga de la aplicación podremos seguir recibiendo y gestionando dicha información de manera que seremos capaces de ofrecer a los usuarios información actualizada en todo momento.

Para gestionar el proceso de conexión con el servicio y de recepción de mensajes desde este, se ha creado la clase *GestorMensajes* que usaremos para categorizar los mensajes que recibamos desde el servicio en la actividad *MainActicity.*

7.3.1 Main Activity

Esta actividad es la más importante de toda la aplicación, en ella se ha implementado la funcionalidad necesaria para poder mostrar al usuario la lista con los recursos y áreas geolocalizadas próximas a su posición, como ya se ha comentado este proceso es automático para lo cual esta actividad hace uso de los servicios *ServicioSegundoPlanoMQTT* y *ServicioImagenes* . Además es esta actividad la que, al recibir la lista con los recursos próximos al usuario crea una notificación para avisar a este del hallazgo por lo que también hace uso del servicio *GestorNotificaciones.*

Vemos primero como se inicializa la actividad, para ello debemos implementar el método *onCreate* que, como vimos en el apartado dedicado al ciclo de vida de las actividades, se llamará en el momento en que el sistema Android cree la actividad. En este método debemos llevar a cabo la inicialización de los servicios que va a utilizar la actividad además de establecer la interfaz que deberá mostrarse al usuario. Para ello, dentro del método *onCreate* realizamos las siguientes operaciones:

En primer lugar comprobamos si tenemos el nombre del usuario, de no ser así abrimos la actividad *AjustesActivity* desde la cual el usuario puede introducir el nombre de usuario que haya utilizado en la aplicación *Owntracks*. Este dato es fundamental por lo que es lo primero que comprobamos al iniciar la actividad. El valor actual del nombre del usuario lo obtendremos usando un objeto de tipo *SharedPreferences* gracias al cual podremos guardar en el dispositivo pares clave-valor para posteriormente poder recuperarlos. Usando este objeto recuperamos el nombre del usuario de la siguiente manera:

```
prefs = getSharedPreferences("UserPreferences",
                             Context.MODE_PRIVATE);
username =
prefs.getString("username",getString(R.string.noUsername));
```

Una vez hemos obtenido el nombre del usuario abrimos la actividad *AjustesActivity* si es necesrio:

```
if(username.equals(getString(R.string.noUsername))) {

      Intent intent = new Intent(getApplicationContext(),
                                 AjustesActivity.class);
       intent.addFlags (FLAG_ACTIVITY_SINGLE_TOP);
       startActivity(intent);
}
```

En caso de que si que tengamos el nombre del usuario establecemos que diseño de interfaz debe mostrarse al crear esta actividad, en una aplicación Android este diseño se especifica en un archivo XML llamado *layout* o diseño que contiene la información sobre los diferentes componentes gráficos que formarán la pantalla así como sus propiedades. Para esta actividad usaremos el diseño llamado *activity_main.* Además, ya que queremos que la aplicación tenga una barra de herramientas en la parte superior de la pantalla debemos especificarlo en este momento.

```
setContentView(R.layout.activity_main);
Toolbar myToolbar = (Toolbar) findViewById(R.id.my_toolbar);
setSupportActionBar(myToolbar);
```

Tras haber definido que diseño de interfaz debe mostrarse en esta actividad inicializamos los servicios que vamos a emplear en la misma, en concreto el *ServicioImagenes* y el *GestorNotificaciones*. Después configuramos el listado de recursos geolocalizados llamada *listaRecursos* en el que se mostrarán tanto los recursos cercanos como las áreas en las que se encuentre el usuario. Para configurar este listado debemos especificar el diseño de los elementos que formarán parte del mismo para lo que usaremos un objeto de tipo *Adapter* en el que especificaremos el diseño de estos elementos, en nuestro caso el diseño se ha definido en el archivo *item_lista.*

```
adapter = new ItemAdapter(this, R.layout.item_lista,listItems);
listaRecursos.setAdapter(adapter);
```

El último paso para configurar el listado consiste en definir la funcionalidad que queremos que se ejecute cuando un usuario seleccione uno de los elementos del listado, en nuestro caso queremos mostrar una nueva actividad en la que se de información detallada sobre el recurso o el área seleccionada. Para ello primero obtenemos el objeto seleccionado y posteriormente usamos un objeto de tipo *Intent* para pasar la información a la nueva actividad, por último lanzamos la actividad *DetalleActivity* y llamamos al método *onDetroy* que veremos más adelante, a continuación se muestra el proceso descrito:

```
ListItem i = listItems.get(position);

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

intent.putExtra(getString(string.recursos),
      i.getRecurso());

startActivity(intent);
```

Por último en el método *onCreate* actualizamos el valor de una variable booleana llamada *esNuevoUsuario*, como su nombre indica esta variable la usamos para representar si es la primera vez que el usuario utiliza la aplicación.

Esta información es importante ya que como veremos más adelante en caso de que sea un nuevo usuario debemos notificar a su apicación *Owntracks* con la información sobre las áreas geolocalizadas que existan en ese momento en el sistema.

Una vez finalizado el método *onCreate*, se habrá creado la actividad y esta pasará al siguiente estado en su ciclo de vida, en concreto al estado *onStart.* En la

actividad *MainActivity*, este estado es fundamental para que el proceso de actualización de la lista de recursos funcione correctamente. Como se puede observar en la ilustración veintinueve [pág. 85] el método *onStart* se llamará cada vez que el usuario vuelva a esta actividad después de haberla abandonado bien porque ha navegado a otra actividad de la aplicación o bien porque ha salido de la aplicación para utilizar otras funciones de su teléfono.

Independientemente de porqué ha salido de la actividad, siempre que vuelva debemos llevar a cabo ciertas operaciones. La primera de ellas es inicializar el gestor de mensajes, este es un objeto de tipo *GestorMensajes* que extiende la clase Android *BroadcastReceiver,* la implementación de esta clase se discutirá más adelante. Este gestor de mensajes nos sirve para comunicar la actividad con el servicio *ServicioSegundoPlanoMQTT* el cual recibe los mensajes mandados desde el servidor. Para ello debemos crear un nuevo objeto de tipo *GestorMensajes*, indicar a que tipo de mensaje nos queremos suscribir y por último arrancarlo. Los diferentes tipos de mensajes se verán con más detalle en el apartado dedicado al componente *GestorMensajes* [pág. 93].

Para filtrar los mensajes a los que nos queremos suscribir emplearemos un *IntentFilter* y para poner en marcha el gestor de mensajes se utiliza el método *registerReceiver* de la clase *LocalBroadcastManager* como vemos a continuación:

```
gestorMensajes = new GestorMensajes();
IntentFilter filter = new
IntentFilter(getString(string.connection_status_action));
filter.addAction(getString(string.entrada_area_action));
filter.addAction(getString(string.salida_area_action));
filter.addAction(getString(string.recursos_action));
filter.addAction(getString(string.actulizacion_areas_action));
filter.addAction(getString(string.actulizacion_posicion_usuario)
);
LocalBroadcastManager.getInstance(this).registerReceiver(gestorM
ensajes, filter);
```

Tras haber creado el gestor de mensajes debemos inicializar el servicio *ServicioSegundoPlanoMQTT* para que este nos pueda proporcionar la información que deberemos mostrar al usuario sobre los recursos y áreas geolocalizadas. Para ello debemos saber el nombre de usuario que emplea el usuario en la aplicación Owntracks ya que lo necesitaremos para poder suscribirnos a los temas MQTT más adelante para poder recibir y enviar mensajes al servidor,

para iniciar el servicio hacemos una llamada al método *startService* tal como vemos a continuación:

```
Intent i = new Intent(this, ServicioSegundoPlanoMQTT.class);
i.putExtra("username", username);
```

```
startService(i);
```

Nótese el uso de un objeto de tipo *Intent* antes de inicializar el servicio. Como vimos anteriormente este tipo de objetos se utiliza entre otras cosa para inicializar servicios en segundo plano o para pasar datos entre distintas actividades. En este caso estamos pasando al servicio *ServicioSegundoPlanoMQTT* que vamos a inicializar el nombre del usuario.

Tras esto pasamos a comprobar si tenemos información sobre recursos o áreas geolocalizadas, de ser así la mostraremos empleando el método *actualizarDatos* y en caso contrario mostraremos un texto informando al usuario de que no se han encontrado recursos y que la interfaz se actualizará en el momento que se encuentren, también ocultamos el botón para mostrar el mapa con los recursos ya que no tenemos ninguno que mostrar

Recordemos que esta funcionalidad está dentro del método *onStart* de modo que se ejecutará cada vez que se muestre la ventana principal, de este modo conseguimos siempre que el usuario consulte la aplicación tenga los datos del listado de recursos actualizados con los últimos resultados.

Para almacenar la información sobre los recursos y áreas encontradas, se han utilizado dos listas dentro de la clase *MainActicity* por un lado tenemos una lista para almacenar las áreas geolocalizadas y por el otro una lista para el resto de recursos. Estas listas se actualizan automáticamente cada vez que la aplicación recibe desde el servidor nueva información sobre los recursos cercanos, cuando esto ocurre, el servicio *ServicioSegundoPlanoMQTT* notifica a la actividad a través del objeto *GestorMensajes* que veremos en más detalle a continuación.

Por último, en la actividad *Main*Activity también tenemos la funcionalidad para abrir las actividades *MapaActivity* y *AjustesActivity.* La primera actividad mostrará un mapa con los recursos geolocalizados sobreimpresos, para ello una vez el usuario pulsa sobre el botón “Abrir Mapa” se llama al método *onMapButtonClick* en el que para cada recurso geolocalizado creamos un objeto de tipo *MarcadorDTO* que, como vimos en el apartado dedicado al modelo de datos [pág. 87], contendrá la latitud y longitud del recurso así como su descripción. Estos objetos los pasaremos mediante un *Intent* a la nueva actividad para poder crear el mapa y la ejecutaremos mediante el método *startActivity*.

Por otro lado cuando el usuario pulse sobre una de las opciones del menú desplegable que hay en la parte superior derecha de la pantalla se llama al método *onOptionsItemSelected.* Las dos opciones que puede elegir son: abrir la ventana de ajustes o salir completamente de la aplicación cerrando el servicio en segundo plano. Si selecciona la primera se abre la actividad *AjustesActivity*

mientras que en el segundo caso, primero se detiene el servicio *ServicioSegundoPlanoMQTT* usando el método *stopService* y por último se detiene la aplicación llamando al método *finish.*

7.3.2 Gestor de Mensajes

Como acabamos de ver en esta clase se implementa la funcionalidad para poder recibir información desde el *ServicioSegundoPlanoMQTT* y de esta manera poder actualizar la información que mostramos en la actividad MainActivity. El proceso por el cual se envía y se recibe esta información es el siguiente: En primer lugar el servicio *ServicioSegundoPlanoMQTT* difunde la información al resto de componentes de la aplicación pero solo será recibida por aquellos componentes que implementen la interfaz *BroadcastReceiver*, en nuestro caso, el único componente que lo hace es la clase *GestorMensajes*. Por lo tanto, este componente recibirá el mensaje desde el servicio y a su vez será el responsable de actualizar las listas de áreas y recursos geolocalizados en la actividad *MainActivity.* Para ello, una vez ha recibido un mensaje debe clasificarlo según su tipo el cual nos llega en el propio mensaje desde el servicio *ServicioSegundoPlanoMQTT.*

Un mensaje puede ser de uno de los siguientes tipos:

- Entrada en un área geolocalizada
- Salida de un área geolocalizada
- Encontrados nuevos recursos geolocalizados
- Actualización de la posición del usuario
- Actualización
- Cambio en el estado de la conexión con el servidor.

Dependiendo del tipo del mensaje actuaremos de una forma diferente, para los mensajes de tipo “entrada en área” y “recursos localizados” lo que haremos será actualizar la lista de áreas y recursos respectivamente con los nuevos datos que nos lleguen en el mensaje. Si por el contrario el mensaje es de tipo “salida de área” quitaremos la información sobre esa área en concreto de la lista. En caso de que el mensaje sea “actualización de la posición del usuario”, la aplicación guardará la información sobre la posición actual del usuario en un objeto de tipo *PuntoGeográfico* que se usará a la hora de mostrar esta información en la actividad *MapasActivity*.

También puede darse el caso en el que perdamos la conexión con el servidor por algún motivo. En estos casos mostraremos un pequeño mensaje en pantalla avisando de que se ha perdido la conexión con el servidor y, una vez recuperada esta, mostraremos otro mensaje para indicar al usuario que la conexión se ha restablecido.

7.3.3 Servicio Imagenes

Este servicio se encarga de descargar las imágenes asociadas a un recurso, debido a que esta operación es bastante lenta y podría llevar a que la aplicación se detuviera durante unos segundos mientras se procesa la descarga, se ha decido que la operación se desarrolle de forma asíncrona en un hilo diferente al que controla la interfaz de la aplicación. Para ello se ha utilizado la clase *AsyncTask* de *Android* que nos proporciona la funcionalidad necesaria para conseguir el comportamiento asíncrono deseado.

Para implementar nuestra tarea asíncrona se ha creado este servicio, en el se define el método *cargarImagen*, el cual recibe como argumentos el componente de la interfaz en el que hay que cargar la imagen (I*mageView*) y la url desde la que descargar la imagen deseada. Este método creará una nueva instancia de la clase *DescargarImagen* que es nuestra tarea asíncrona.

En esta clase que extiende de *AsyncTask* se han implementado tres métodos: *doInBackground* y *onPostExecute.* El primero de ellos es donde se llevará a cabo el proceso de descarga de la imagen y, una vez terminado, se llamará al segundo método que es el que asignará la imagen al componente *ImageView* el cual la mostrará en la interfaz.

Para gestionar la descarga de la imagen, el método *doInBackground* utiliza un objeto de tipo *InputStream* para descargar la imagen y posteriormente transforma el contenido de este objeto en un *Bitmap* que es lo que finalmente devuelve este método. Si por algún motivo se produjera algún error al descargar la imagen, como por ejemplo que el servidor donde esté alojada no esté disponible o que haya sido borrada del mismo, el método devolverá un icono en sustitución de la imagen.

```
Bitmap mIcon11 =
BitmapFactory.decodeResource(context.getResources(),
                    R.drawable.page_not_found);
```

```
try {
   InputStream in = new java.net.URL(urldisplay).openStream();
   mIcon11 = BitmapFactory.decodeStream(in);
   in.close();
} catch (Exception e) {
   Log.e("Error", e.getMessage());
   e.printStackTrace();
}

return mIcon11;
```

Por último una vez se ha terminado de descargar la imagen se llama al método *onPostExecute* que recibe como parámetro un objeto de tipo *Bitmap* que usaremos para pasar esta imagen al componente *ImageView.* Este método es llamado por *Android* de forma automática una vez ha finalizado el método *doInBackground.*

7.3.4 Servicio Segundo Plano MQTT

Como acabamos de ver, el componente *ServicioSegundoPlanoMQTT* juega un papel fundamental en nuestra aplicación ya que nos permite ser capaces de actualizar el conjunto de recursos y áreas geolocalizadas próximas al usuario aunque la aplicación esté detenida.

Para lograr esto, esta clase hereda de la clase *Service* que nos proporciona Android gracias a la cual podemos hacer que el servicio se ejecute de forma indefinida en segunda plano aunque el usuario deje de utilizar la aplicación. Esta clase nos proporciona una serie de métodos para poder gestionar nuestro servicio, el más importante de los cuales es el método *onStartCommand* en el que deberemos implementar la funcionalidad que queremos llevar a cabo al inicializar el servicio. En nuestro caso, queremos que el servicio se inicie en segundo plano para no bloquear al resto de la aplicación para ello debemos realizar una llamada al método *startForeground* al cual deberemos pasarle un objeto de tipo *Notification*. Esto es debido a que Android nos obliga a notificar al usuario que vamos a iniciar un servicio en segundo plano, de esta manera, mientras el servicio esté activo el usuario verá la notificación en la parte superior de sus dispositivo.

Una vez hemos iniciado en segundo plano el servicio debemos iniciar el *ServicioMQTT* que igual que en el servidor será la clase responsable de recibir y enviar los mensajes MQTT. Al iniciar este servicio, le pasaremos una referencia al *ServicioSegundoPlanoMQTT* para que cada vez que nos llegue un nuevo mensaje con información sobre nuevos recursos geolocalizados, el *ServicioMQTT* nos haga llegar esta información y, posteriormente, se la pasaremos a la actividad

MainActivity usando la funcionalidades que nos proporciona la clase *LocalBroadcastManager* que nos permitirá comunicarnos con la actividad a través del componente *GestorMensajes.*

Para finalizar la inicialización del servicio *ServicioSegundoPlanoMQTT,* una vez hemos iniciado el componente *ServicioMQTT,* enviamos información sobre las áreas geolocalizadas en las que ha entrado en usuario así como de los recursos geolocalizados próximos a este utilizando el método *enviarUltimosDatos*. Para ello, *ServicioSegundoPlanoMQTT* dispone una lista de áreas y otra de recursos que se actualizan conforme el componente *ServicioMQTT* recibe mensajes desde el servidor. De esta manera disponemos siempre de información actualizada sobre las áreas a las que ha entrado el usuario así como de aquellos recursos que se encentran cerca de el.

Hay que recordar que toda la operación de inicialización del servicio se lleva a cabo en el momento en que la actividad principal *MainActivity* se inicia por lo que esta recibirá los últimos datos disponibles para poder actualizar la interfaz según corresponda por ejemplo, tras un periodo de inactividad en el cual el usuario ha cerrado la aplicación, en el momento en el que la vuelva a abrir recibirá la información actualizada sobre los recursos cercanos y las áreas a las que haya accedido mientras no usaba la aplicación.

Cuando se reciba un mensaje desde el servidor a través del componente *ServicioMQTT* este invocará a una serie de métodos del servicio en segundo plano que serán los que notifiquen a la actividad *MainActivity* con la información recibida, para ello se han creado seis métodos los cuales se describen a continuación:

- ***notificarEntradaArea:*** Este método recibirá un objeto de tipo *RecursoGeolocalizadoDTO* y notificará a la actividad *MainActivity* que el usuario ha entrado en un área. Para ello enviaremos un objeto de tipo *intent* en el que además de los datos del área especificaremos que el usuario está entrado en un área. Como vimos al tratar la implementación de *MainActivity* el tipo de movimiento es importante par la correcta gestión de los mensajes.

  Además de notificar la entrada en el área, nos guardamos el objeto de tipo *RecursoGeolocalizadoDTO* en una lista que nos será útil en caso de que debamos enviar a la actividad *MainActivity* información sobre todos las áreas en las que hemos entrado.

  ```
  public void notificarEntradaArea(RecursoGeolocalizadoDTO area)
  {
  ```

```
        Intent intent = new Intent();
        intent.setAction(getString(R.string.entrada_area_action));
        intent.putExtra( getString(R.string.area),area);
        localBroadcastManager.sendBroadcast(intent);
        if(!listaAreas.contains(area)){
            listaAreas.add(area);
        }
    }
```

- ***notificarSalidaArea***: Este método es muy similar al anterior, la única diferencia es que el tipo de movimiento ahora es de salida y que en lugar de guardar la información sobre el área en una lista, lo que haremos será elimina la información que tengamos almacenada sobre esa área ya que a partir del momento en que salimos de ella, ya no vamos a mostrar información sobre área al usuario.

```
intent.setAction(getString(R.string.salida_area_action));
intent.putExtra( getString(R.string.area),area);
localBroadcastManager.sendBroadcast(intent);
listaAreas.remove(area);
```

- ***notificarRecursos***: Este método envía una lista de recursos geolocalizados a la actividad *MainActivity* de forma análoga a como hemos visto en el método *notificarEntradaArea*, además guardaremos la lista con los recursos por si hay que enviar esta información a *MainAactivity* más adelante.

- ***enviarUltimosDatos***: Método que usaremos para enviar tanto la lista de áreas como la de recursos geolocalizados. Esta funcionalidad es fundamental ya que nos permite refrescar los datos cada vez que el usuario vuelve a la aplicación, para ello llamaremos a este método desde el servicio *ServicioMQTT* cada vez que nos conectemos al *broker*. Debido a que el servicio en segundo plano no se detiene aunque el usuario salga de la aplicación siempre dispone de la información actualizada de manera que cuando la aplicación vuelve a activarse e intenta conectarse de nuevo con el *broker* invocamos este método y enviamos a la actividad *MainActivity* las listas de áreas y recursos geolocalizados actualizados.

- ***notificarPosicionUsuario***: Usaremos este método para poder actualizar la posición actual en la que se encuentra el usuario de cara a poder representarla en el mapa que mostramos en la actividad *MapasActivity,* para ello recibimos un objeto de tipo *PuntoGeografico* y creamos un *Intent*

Por último debemos tener en cuenta que al tratarse de un servicio que se ejecuta en segundo plano de forma indefinida, debemos implementar la funcionalidad para poder detener su ejecución. Este proceso consta de dos partes, en primer lugar el usuario puede finalizar la ejecución de la aplicación seleccionando la opción "Salir" en el menú desplegable ubicado en la esquina superior derecha de la aplicación. Una vez seleccionada esta opción, en la actividad *ActivityMain* se realizará una llamada al método *stopService* donde indicaremos que queremos detener el servicio *ServicioSegundoPlanoMQTT*. Esto hará que se llame al método *onDestroy* en el servicio y, dentro de este utilizaremos el método *stopForeground* para detener la ejecución en segundo plano del servicio y para quitar la notificación que indicaba al usuario que el servicio estaba ejecutándose.

7.3.5 Servicio MQTT

A continuación vamos a comentar brevemente algunos de los detalles de la implementación de la clase *ServicioMQTT* en los que difiere del servicio visto en el apartado dedicado a la implementación del servidor [pág. 76]. Como hemos dicho anteriormente, este servicio utiliza los métodos descritos en el apartado anterior dedicado al *ServicioSegundoPlanoMQTT*.

La principal diferencia entre este servicio y el implementado en el servidor reside en los temas a los que se suscribirá el servicio para poder recibir desde el servidor los recursos geolocalizados cercanos, las áreas a las que entre el usuario así como los cambios de posición que realice.

Estos *topics* son los siguientes:

- *TOPIC_ENTRADA_AREA* = "info/"+ USUARIO_OWNTRACKS +"/enter"
- *TOPIC_SALIDA_AREA* = "info/"+ USUARIO_OWNTRACKS +"/exit"
- *TOPIC_RECURSOS* = "info/"+ USUARIO_OWNTRACKS +"/resources"
- *TOPIC_POSICION_USUARIO* = "owntracks/"+ USUARIO_OWNTRACKS +"/+"
- *TOPIC_NUEVO_USUARIO* = "owntracks/newuser"

El tema *TOPIC_ENTRADA_AREA* lo usaremos para recibir mensajes que contendrán el área geolocalizada en la que ha entrado el usuario, por el contrario, para recibir mensajes cuando el usuario salga de un área se publicará un mensaje en el tema *TOPIC_SALIDA_AREA*. Conforme cambie la posición del usuario, estos

cambios se publicarán en el tema *TOPIC_POSICION_USUARIO* y, los recursos geolocalizados próximos se publicarán en el tema *TOPIC_RECURSOS.*

El parámetro "USUARIO_OWNTRACKS" nos llegará desde la actividad *MainActivity* y se corresponderá con el nombre que el usuario haya introducido en la ventana de ajustes de la aplicación, de esta manera cada aplicación se suscribirá para recibir información sobre la posición y los recursos cercanos de un usuario de Owntracks.

La conexión con el broker *MQTT* se realiza de manera idéntica a como se hace en el servidor con el cambio de los temas a los que nos vamos a suscribir que en este caso serán los que acabamos de ver. Una vez conectados ya pueden comenzar a llegarnos mensajes *MQTT*, en el método *messageArrived* se filtrará el tema en el que se ha publicado el mensaje y se procesará según corresponda.

Si el tema es *TOPIC_ENTRADA_AREA* o *TOPIC_SALIDA_AREA,* lo primero que hacemos es obtener un objeto de tipo *RecursoGeolocalizadoDTO* a partir de los datos del mensaje, para ello empleamos los métodos que nos proporciona la clase *ServicioJSON.* Una vez obtenido este objeto, usamos el servicio *ServicioSegundoPlanoMQTT* para notificar la entrada o salida del área geolocalizada como ya vimos en el apartado dedicado a este componente se utilizan los métodos *notificarEntradaArea* y *notificarSalidaArea.*

Cuando el tema en el que se publica el mensaje es *TOPIC_RECURSOS* utilizamos de nuevo la clase *ServicioJSON* para obtener la lista de recursos con lo que obtenemos una lista de objetos de tipo *RecursoGeolocalizadoDTO* que se la pasamos al servicio *ServicioSegundoPlanoMQTT* para poder cambiar la interfaz de usuario mostrando los nuevos recursos usando el método *notificarRecursos.*

En caso de que el tema del mensaje recibido sea *TOPIC_POSICION_USUARIO* obtendremos un objeto *PuntoGeograficoDTO* a partir del contenido del mensaje, este objeto contendrá la longitud y la latitud de la posición actual del usuario. Cuando el usuario quiera ver en el mapa la posición de los recursos geolocalizados que tenga cerca, usaremos este objeto para poder indicar sobre el mapa la posición del usuario. Para mantener actualizada la información sobre la posición del usuario se utiliza el método *notificarPosicionUsuario* de la clase *ServicioSegundoPlanoMQTT.*

7.3.6 Servicio JSON

Para finalizar solo nos queda comentar brevemente los detalles de la implementación de esta clase la cual se encarga de transformar el contenido JSON

de los mensajes que nos llegan a la aplicación desde el servidor en objetos que podremos usar en la misma.

En la aplicación desarrollada necesitamos llevar a cabo tres transformaciones, es necesario obtener a partir de texto en formato JOSN objetos de tipo *PuntoGeograficoDTO y RecursoGeolocalizadoDTO*. Para ello disponemos de los siguientes métodos:

- *parseRecursos*: Recibe como parámetro una cadena de caracteres y devuelve una lista formada por objetos de tipo *RecursoGeolocalizadoDTO.*

- *parsePuntoGeografico*: Recibe como parámetro una cadena de caracteres y devuelve un objeto de tipo *PuntoGeograficoDTO*.

- *parseArea:* Recibe como parámetro una cadena de caracteres y transforma esta información en un nuevo objeto de tipo *RecursoGeolocalizadoDTO,* para ello emplea el método anterior y después establece la propiedad *esArea* a true, de manera que podamos identificar a este objeto como un área geolocalizada.

La implementación de estos métodos es similar a los de la clase *ServicioJSON* [pág. 71] del servidor por lo que también utilizaremos la librería *Gson* para poder extraer la información de los mensajes en formato JSON y transformarlos en los objetos del modelo requeridos, por ello no entraremos en este apartado en más detalles sobre la implementación de estos métodos.

7.3.7 Detalle Activity

Esta actividad es la encargada de controlar la ventana en la que se muestra el detalle con la información de un recurso o área geolocalizada [pág. 56]. La actividad recibirá desde *MainActivity* el el recurso seleccionado por el usuario, extraerá la información e inicializará la interfaz con los datos correspondientes.

Todo esto se lleva a cabo en el método *onCreate*, primero obtenemos varias referencias a los elementos que conforman la interfaz para posteriormente poder asignar a estos elementos los datos del recurso seleccionado. Después obtenemos el objeto *RecursoGeolocalizadoDTO* seleccionado, para ello accedemos al objeto *Intent* que contiene los datos que recibe la actividad.

De este objeto podemos extraer la información sobre el área o el recurso geolocalizado, en concreto las imágenes asociadas al objeto y la información asociada al mismo de manera que podemos actualizar el valor de los campos de

la interfaz que obtuvimos anteriormente con los datos del área o recurso seleccionado.

```
RecursoGeolocalizadoDTO recurso =
getIntent().getParcelableExtra(getString(R.string.recursos));

if(recurso != null){
      nombre.setText(recurso.getNombre());
      info.setText(recurso.getInfo());
}
```

Finalmente configuramos el carrusel de imágenes, descargando cada una de las imágenes asociadas al recurso o área, para ello emplearemos el servicio *ServicioImagenes*.

7.3.8 Mapas Activity

La actividad *MapasActivity* se ocupa de mostrar una serie de recursos geolocalizados sobre un mapa de manera que para el usuario sea más fácil ver donde están situados los recursos que tiene cerca.

A esta actividad se accede desde la actividad *MainActivity* al pulsar el botón correspondiente. A la hora de pasar los datos sobre los recursos geolocalizados desde la actividad principal a esta se usará una lista de objetos de tipo *MarcadorDTO*. Estos objetos contendrán la latitud, longitud y descripción de cada uno de los recursos cercanos al usuario en el momento de abrir el mapa. Además también recibiremos un objeto *MarcadorDTO* con la posición actual del usuario para poder indicarla en el mapa.

El mapa que vamos a utilizar nos lo proporciona Android y se trata del mismo mapa que podemos ver al utilizar la aplicación *Google Maps*. Para que el mapa se ajuste mejor a nuestras necesidades realizaremos algunos cambios, por ejemplo, usaremos el mapa de tipo satélite que nos permite ver el terreno en lugar del mapa por defecto, también centraremos la vista sobre la posición actual del usuario y por último crearemos los marcadores para poder indicar la posición de los recursos geolocalizados. Para llevar a cabo esta última operación deberemos recorrer la lista de recursos que nos llega desde la actividad *MainActivity* y, para cada uno de ellos, crearemos un marcador en el mapa. Estos marcadores están compuestos por un icono que se mostrará en una posición determinada por la latitud y la longitud del recurso geolocalizado que represente el marcador. Además cada marcador contendrá información sobre el recurso que representa para que cando el usuario pulse sobre el se muestre en el mapa una breve

descripción del recurso así como la distancia que lo separa de la posición actual del usuario.

7.3.9 Ajustes Activity

Esta actividad se encarga de gestionar la ventana desde la que el usuario puede modificar varios parámetros de la aplicación, en concreto, puede cambiar el nombre de usuario que utiliza en la aplicación Owntracks así como especificar si quiere que se le muestren notificaciones cuando se encuentren nuevo recursos geolocalizados.

Para poder guardar los ajustes que el usuario seleccione utilizaremos el objeto *SharedPreferences* como ya vimos en la implementación de la actividad MainActivity. En nuestro caso vamos a guardar los dos datos mencionados anteriormente, por un lado el nombre del usuario para poder utilizarlo en la actividad *MainActivity* a la hora de inicializar el servicio *ServicioSegundoPlanoMQTT* para suscribirnos a los temas necesarios para recibir desde el servidor las actualizaciones de recursos y áreas cercanas. También guardaremos usando este sistema un valor que nos indique si el usuario quiere o no que se le muestren las actualizaciones que genera la aplicación.

Esta actividad se comportará de forma diferente dependiendo de si es la primera vez que el usuario entra en ella, es decir es un nuevo usuario usando por primera vez la aplicación, o si ya ha entrado previamente. En caso de que sea la primera vez que se accede la actividad permite al usuario introducir su nombre de usuario en la aplicación *Owntracks* que como hemos visto se trata de un dato fundamental para el correcto funcionamiento de la solución. Por el contrario si ya introdujo su nombre anteriormente se le da la opción de editarlo así como de configurar si quiere que la aplicación le muestre notificaciones o no.

Para ello, en esta actividad cagaremos los datos que tenemos guardados al abrir la ventana de ajustes y, cuando el usuario pulse el botón para guardar los cambios comprobaremos que estos son correctos y los guardaremos en el dispositivo.

En el método *onCreate* obtendremos una referencia al objeto *SharedPreferences* para después obtener el nombre de usuario guardado en el dispositivo, tras esto comprobamos si el nombre de usuario existe, en caso de no existir al recuperarlo del objeto *SharedPreferences* este nos devolverá el valor *"noUsername"* ya que este es el valor por defecto que hemos indicado al llamar al método *getString*. Por lo tanto si aún no existe un nombre de usuario mostramos el diseño de la ventana llamado *welcomeLayout* que permitirá al usuario introducir este dato, si ya existe un nombre de usuario se le mostrará el diseño *settingsLayout* el cual le permitirá modificar su nombre de usuario así como activar y desactivar las notificaciones de la aplicación. Por último actualizamos la variable booleana *esNuevoUsuario* que usaremos a la hora de guardar los cambios.

```
prefs = getSharedPreferences("UserPreferences",
Context.MODE_PRIVATE);
String username =
prefs.getString("nombreUsuario",getString(R.string.noUsername));

 if(username == getString(R.string.noUsername)){
      settingsLayout.setVisibility(View.INVISIBLE);
      welcomeLayout.setVisibility(View.VISIBLE);
      esNuevoUsuario = Boolean.TRUE;

      newUsernameField.setText("");
 }else{
      settingsLayout.setVisibility(View.VISIBLE);
      welcomeLayout.setVisibility(View.INVISIBLE);
      usernameField.setText(username);

      text.setText("Nombre de usuario Owntracks");
 }
```

Una vez el usuario ha realizado los cambios que considere oportuno, puede pulsar el botón "Guardar cambios" el cual llamará al método *guardarCambios* en el que comprobaremos que se haya introducido un nombre de usuario, si no lo estuviera mostraríamos un mensaje para que el usuario sepa que debe introducir este dato. Si por el contrario el nombre de usuario contiene un valor válido guardamos los datos utilizando el objeto *SharedPreferences.*

```
SharedPreferences.Editor editor = prefs.edit();
editor.putString("username", nombreUsuario);
editor.putBoolean("notificaciones", notificaciones.isChecked());
editor.apply();
```

Tras guardar los cambios, le indicaremos al usuario que los cambios se han guardado correctamente y a continuación se comprueba si estamos guardando

un nuevo usuario o editando uno existente, esto es importante ya que si se trata del primer caso debemos informar a la actividad *MainActivity* de que el usuario ha introducido su nombre por primera vez, ya que como vimos en el apartado dedicado a esta actividad cuando esto ocurre hay que llevar a cabo una serie de operaciones para refrescar las áreas geolocalizadas en la aplicación *Owntracks.* Para saber si el usuario es nuevo o no nos fijaremos en el valor de la variable *esNuevoUsuario*, si estamos ante el caso de que el usuario es nuevo entonces debemos añadir esta información al *Intent*.

```
Intent intent = new Intent(getApplicationContext(),
                           MainActivity.class);
if(esNuevoUsuario){
      intent.putExtra(getString(R.string.esNuevoUsuario),
                      esNuevoUsuario);
}
```

8. Guía de uso

En este apartado se va a exponer como configurar, desplegar y utilizar la solución propuesta a lo largo del presente documento de modo que sirva a modo de guía para los posibles usuarios. Esta guía va a estar dividida en varios apartados, en primer lugar veremos como configurar un *broker* de comunicaciones *MQTT* y como configurar correctamente el servidor para poder acceder a este. Posteriormente se explicará como configurar la aplicación *Owntracks* y, por último hablaremos de la configuración de la aplicación móvil.

Una vez tengamos todo configurado pasaremos a ver un ejemplo de uso de la solución mostrando como se actualiza la información en nuestro teléfono móvil así como las diferentes interacciones que podemos llevar a cabo con la aplicación.

8.1 Configuración del servidor y del gestor de comunicaciones

Comencemos viendo como podemos configurar un *broker* de comunicaciones, a la hora de hacerlo tenemos varias opciones disponibles: podemos optar por implementar uno nosotros mismos o por usar uno ya existente. En nuestro caso se ha optado por la segunda opción, en concreto se ha empleado el servicio *CloudMQTT.* Una vez hemos accedido a su página web nos podemos registrar y crear una nueva instancia, esta será nuestro gestor de comunicaciones.

Al crear la instancia se nos muestra la información de la misma como el servidor al que tendremos que conectarnos para usarla, el puerto, el nombre de usuario y de contraseña necesarios para poder conectarnos a ella. Estos datos nos harán falta más adelante cuando configuremos el servidor. Una vez creada la instancia ya podríamos arrancarla y estaría lista para usarse.

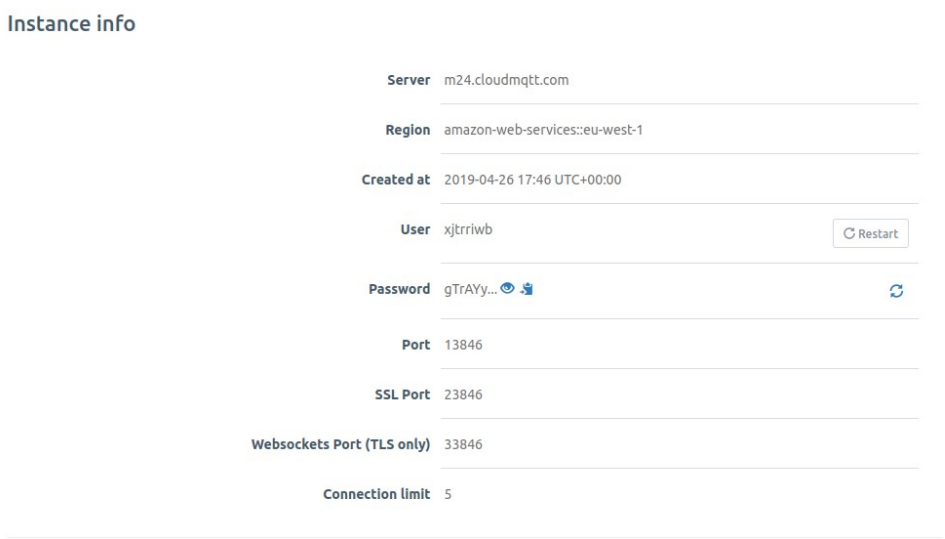

Ilustración 25: Información de la instancia creada con CloudMQTT. Fuente: elaboración propia

Tras haber configurado el *broker* podemos pasar a configurar el servidor. Para ello en primer lugar configuraremos la base de datos *SQLite,* comenzaremos descargando la herramienta *DB Browser for SQLite* esta herramienta nos permitirá acceder a nuestra base de datos y modificar los datos presentes en la misma de manera que podamos introducir los recursos y áreas geolocalizadas que consideremos necesarias. La herramienta la podemos descargar desde la siguiente dirección web: https://sqlitebrowser.org/dl/ en la que tenemos disponibles versiones para varios sistemas operativos. No es necesario instalar esta herramienta en la misma máquina en la que se va a ejecutar el servidor, si esta no dispone de salida gráfica podemos instalar la herramienta en cualquier otra máquina.

Una vez descargada la aplicación *DB Browser for SQLite* la ejecutamos y abrimos el archivo de base de datos que se suministra junto con la solución. Al abrirlo podemos ver las tablas que forman nuestra base de datos, a la hora de introducir datos en ellas solamente debemos trabajar sobre las tablas *areas_geolocalizadas* y *recursos_geolocalizados.* Comenzaremos editando la primera tabla, para ello iremos a la pestaña “Navegar datos” y en el desplegable seleccionaremos la tabla *areas_geolocalizadas.* Se nos mostrarán todos los datos que existan en esta tabla, como es la primera vez que la editamos estará vacía. Podemos añadir nuevas áreas geolocalizadas pulsando el botón “Nuevo Registro” con lo que se creará un registro en blanco que podemos rellenar con los datos que queramos. Los datos que debemos añadir para cada área son: el nombre, la información relacionada con el área, la *url* donde reside la imagen del área, la latitud y la longitud de su centro y por último el radio del área.

Una vez hemos añadido los datos sobre las áreas geolocalizadas guardamos los cambios pulsando el botón “Guardar Cambios”.

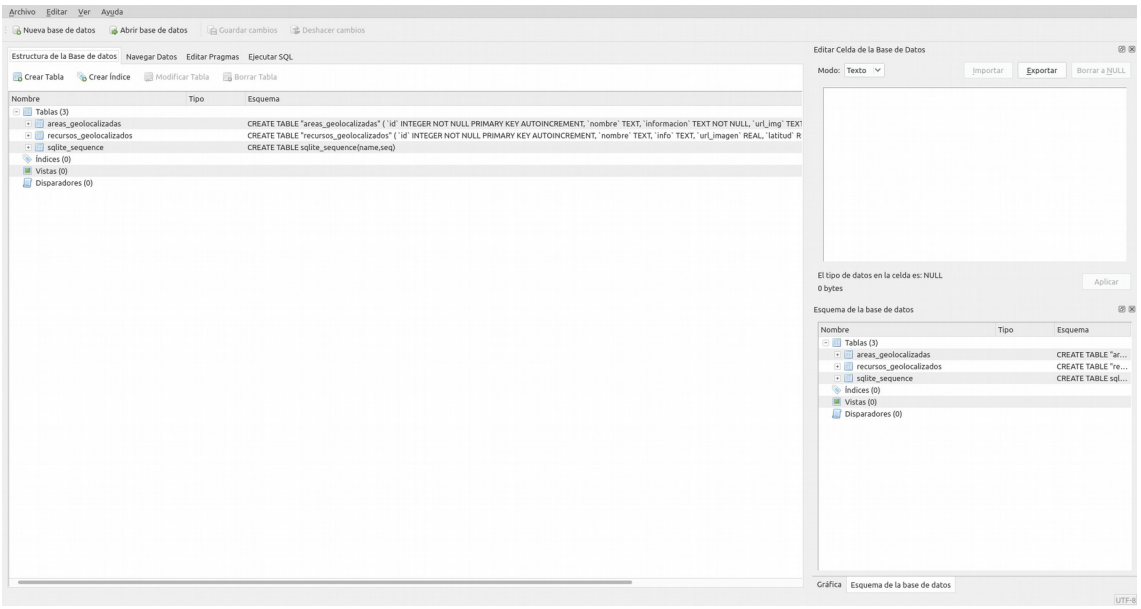

Ilustración 26: Captura de la apliación DB Browser. Fuente: elaboración propia

Con las áreas geolocalizadas creadas podemos crear los recursos geolocalizados modificando la tabla *recursos_geolocalizados.* En esta tabla procederemos de la misma forma que en la anterior creando tantos registros como recursos queramos tener, en cada uno de los recurso debemos rellenar el nombre, la información del recurso, la *url* de la imagen representativa, la latitud, la longitud y por último el identificador del área geolocalizada en la que se encuentra el recurso, si no está dentro de ningún área definida esta columna se puede dejar sin valor.

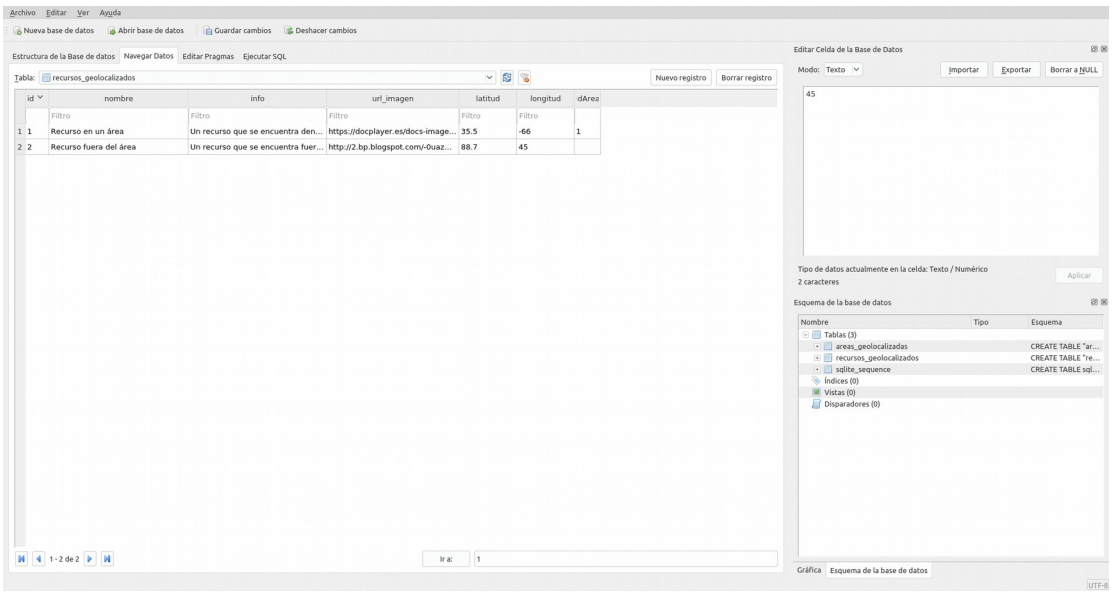

Ilustración 27: Aplicación DB Browser. Fuente elaboración propia

Tras haber añadido todos los recursos y áreas que deseemos tenemos que configurar el servidor, para ello únicamente tenemos que modificar el archivo config.*properties* en el que especificaremos los datos de configuración necesarios para que el servidor funcione correctamente. Este fichero funciona mediante pares clave-valor con los que especificaremos propiedades y su valor, el fichero de configuración ya tiene las propiedades necesarias definidas por lo que solo tenemos que cambiar su valor por el apropiado en nuestro caso.

Las propiedades que debemos especificar son las siguientes:

- ruta_base_datos: En esta propiedad debemos especificar la ruta en nuestro sistema en la que se encuentra el archivo de *SQLite* con nuestra base de datos, por ejemplo esta ruta en un sistema Linux puede ser: "/home/db/.db"
- broker_url: La *URL* en la que podemos acceder a nuestro *broker*
- broker_usuario: El nombre de usuario con el que debemos conectarnos al *broker.*
- broker_contrasena: La contraseña necesaria para poder establecer la conexión con el gestor de comunicaciones.
- crt_file_path: LA ruta al archivo *.crt* del certificado electrónico que se usará para encriptar la comunicación entre el broker y el servidor mediante *SSL*.

Como hemos visto anteriormente, en el caso de utilizar *CloudMQTT*, la información necesaria para estas últimas propiedades se encuentra disponible en el apartado dedicado a la información sobre la instancia.

Para poder iniciar el servidor únicamente debemos ejecutar el siguiente comando desde una terminal en el directorio donde tengamos el ejecutable (fichero con extensión *.jar*) del servidor: "java –jar RUTA_FICHERO_JAR RUTA_FICHERO_PROPERTIES"

8.2 Configuración de las aplicaciones móviles

Una vez hemos configurado y arrancado el servidor veamos los pasos a seguir para configurar las dos aplicaciones móviles que deberemos emplear para poder utilizar la solución presentada en este documento: *Owntracks* y la aplicación desarrollada para este proyecto. Primero debemos descargar la aplicación *Owntracks* desde la tienda de aplicaciones de nuestro dispositivo, una vez instalada podremos configurar los parámetros de conexión para permitir que se

conecte al *broker* que configuramos en el apartado anterior. Desde la aplicación *Owntracks* podemos acceder a la configuración pulsando en la esquina superior izquierda y después sobre la opción “Preferencias”, una vez dentro de este menú deberemos acceder al apartado “Conexión” y elegir el modo “MQTT”. Dentro del menú “Host” introduciremos la *url* del broker y el puerto. Aceptamos los cambios y entramos en el menú “Identificación”, deberemos introducir el nombre de usuario y la contraseña del *broker* así como un identificador para nuestro dispositivo en la opción “ID de Dispositivo”. Por último en la opción “Seguridad” podemos activar el uso del protocolo *TLS* para aumentar la seguridad de las comunicaciones entre *Owntracks* y el gestor de comunicaciones. Si lo hacemos tendremos que indicar el certificado que deseamos emplear.

Tras esto *Owntracks* ya se encuentra correctamente configurado, a partir de ahora solo tenemos que seleccionar uno de los modos de monitorización de los que dispone la aplicación, durante el uso de la solución no es necesario interactuar más con esta aplicación.

Por último solo falta configurar la aplicación móvil, este último paso consiste únicamente en ejecutar la aplicación e introducir en el menú de ajustes el nombre que hayamos elegido al configurar *Owntracks* en el apartado “ID de dispositivo”. Si es la primera vez que utilizamos la aplicación el menú de ajustes se abrirá de forma automática, si por el contrario ya la hemos usado antes podremos acceder a el desde el menú en la esquina superior izquierda de la pantalla.

8.3 Manejo de la aplicación

En este último apartado de la guía de uso se va a mostrar como utilizar la aplicación móvil. A la hora de utilizarla la interacción que debe llevar a cabo el usuario con la aplicación es mínima pero aún así existen ciertas acciones que el usuario puede llevar a cabo.

La ventana principal de la aplicación está dividida en tres secciones, en la parte superior se encuentra una barra con un texto descriptivo de la aplicación y un menú desplegable desde el que acceder a la pantalla de ajustes y cerrar la aplicación. En la zona central de la ventana se muestra el listado de recursos y áreas geolocalizadas que se actualizará automáticamente y, por último, en la zona inferior de la ventana se localiza el botón “Ver Mapa” que abre la vista del mapa donde se muestran los recursos geolocalizados.

Ilustración 28: Captura de la ventana principal de la aplicación antes de tener datos

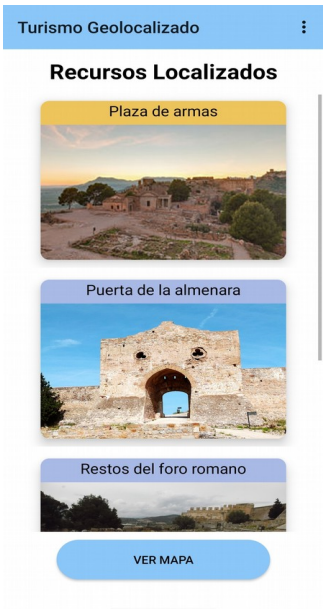

Ilustración 29: Captura de la ventana principal de la aplicación una vez se han recibido datos

Ilustración 30: Captura de la ventana del detalle de un recurso

Cuando se detecta que se ha entrado en un área o que hay recurso próximos, estos se mostrarán en el listado. Las áreas geolocalizadas tiene el nombre sobre un fondo amarillo mientras que los recursos lo tienen sobre uno de color azul. Al pulsar sobre uno de estos elementos se muestra la ventana del detalle, en ella se puede ver la información completa del recurso o área seleccionada. Además, en la parte superior hay un carrusel de imágenes que permite ver varias imágenes relacionadas con el recurso.

Por último vamos a ver las ventanas de ajustes y la del mapa. En la primera se pueden activar o desactivar las notificaciones que genera la aplicación y cambiar el nombre de dispositivo desde la ventana de ajustes, para acceder a ella debemos pulsar sobre el menú desplegable de la barra en la parte superior de la ventana principal de la aplicación.

En la ventana del mapa se muestra la localización de los recursos geolocalizados sobre un mapa de *GoogleMaps*, como ya se ha comentado a esta ventana se accede desde la ventana principal pulsando sobre el botón “Ver Mapa”.

Ilustración 31: Captura de la ventana con el mapa y los recursos

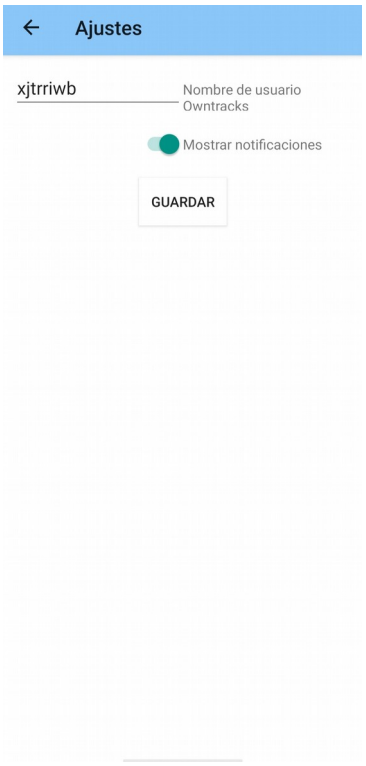

Ilustración 32: Captura de la ventana de ajustes

9. Conclusiones y trabajos futuros

Como conclusiones finales de este proyecto podemos destacar que se ha conseguido crear un prototipo de plataforma para el castillo de Sagunto que posibilita la interacción con el mundo físico a través de la geolocalización de diversos recursos y espacios físicos capaz de ofrecer información en tiempo real adaptándose a la posición de los usuarios explorando el uso de tecnologías como *MQTT* y aplicando, en la medida de lo posible, los principios de la inteligencia ambiental.

Además la solución desarrollada es fácil de mantener y expandir en un futuro gracias a la aproximación modular y los patrones de diseño que se han seguido a la hora de desarrollar tanto la aplicación móvil como la aplicación del servidor y que permiten realizar modificaciones de una forma más sencilla y rápida.

Esta adaptabilidad permite que la plataforma se pueda utilizar en muchos entornos diferentes, en este proyecto en concreto se ha focalizado en un uso turístico de la misma peo también podría aplicarse en otros escenarios como por ejemplo en tiendas u otros establecimientos para orientar a los clientes.

Por otro lado estamos ante un desarrollo cuyo coste de operación es prácticamente cero gracias al uso de tecnologías y estándares no propietarios como *SQLite* o *Owntracks* y a la poca inversión en infraestructura que es necesario llevar acabo para poner en funcionamiento esta solución. Por todo ello resulta adecuada para entornos en los que se quiere ofrecer la capacidad de interaccionar con diferentes recursos físicos sin que ello suponga una gran inversión.

Uno de los mayores retos de este desarrollo de cara al futuro será el control de los usuarios que utilicen la plataforma así como la gestión de la misma por parte de los administradores.

En cuanto a los usuarios, en este prototipo no se ha implementado ningún tipo de sistema que nos permita identificar de forma única a los usuarios que usen la plataforma, podría darse el caso que varios usuarios utilicen el mismo nombre de usuario creándose situaciones en las que la información que recibirán puede no ser correcta.

Por otro lado el tener que utilizar dos aplicaciones (*Owntracks* y la aplicación móvil desarrollada en este proyecto) puede suponer una molestia o dificultad añadida para algunos usuarios a la hora de emplear la plataforma. Sería recomendable explorar la posibilidad de que la propia aplicación sea capaz de acceder a los datos de geolocalización proporcionados por el GPS del dispositivo móvil y los transmitiera al servidor de manera que no fuera necesario emplear la aplicación *Owntracks* para este propósito.

En cuanto a la facilidad que tendrán los administradores del sistema para modificar los contenidos de la plataforma, es cierto que si bien el editor de base de datos *DBBrowser* es una herramienta que nos permite modificar el contenido de la base de datos de una forma relativamente sencilla puede no ser la más adecuada ya que no proporciona una interfaz especialmente intuitiva y ofrece acceso total a la base de datos permitiendo a los usuarios realizar cambios con el riesgo que esto conlleva para el funcionamiento del sistema. Por lo tanto, otro de los posibles desarrollos futuros podría ser desarrollar algún tipo de aplicación que permitiera la gestión de la base de datos de una forma accesible para todo tipo de usuario y que fuera capaz de limitar mediante algún tipo de sistema de permisos las acciones que los usuarios pueden llevar a cabo respecto a la base de datos. Por ejemplo desarrollando una aplicación web que permita realizar las acciones de inserción, borrado, edición y de consulta desde cualquier navegador permitiendo a los usuario modificar los datos de forma segura y cómoda.

Por último otro de los posibles trabajos futuros podría consistir en modificar la plataforma desarrollada para realizar otros tipos de interacción con objetos geolocalizados en otros ámbitos como por ejemplo, controlar una serie de dispositivos domésticos (luces, electrodomésticos, persianas...) según la posición del usuario dentro de la casa.

10. Bibliografía

- *Diccionario de la lengua española* . 23ª ed., ed. del tricentenario, Real Academia Española, 2014.
- Cook, Diane & Augusto Wrede, Juan & Jakkula, Vikramaditya. (2007). *Review: Ambient intelligence: Technologies, applications, and opportunities. Pervasive and Mobile Computing*.
- Schilit, Bill & Adams, Norman & Want, Roy. (1995). *Context-Aware Computing Applications. Proc. of The IEEE Workshop on Mobile Computing Systems and Applications*. 85 - 90. 10.1109/WMCSA.1994.16.
- ESMARTCITY. *Impulso VLCi: despliegue IoT y extensión del proyecto Valencia Smart City*. 2018.
- Fernández Ruiz, Francisco. *El castillo de Sagunto.* El autor, 1979.
- Weiser, M. (1991).*The Computer for the 21st Century. Scientific American, 94-104*.
- Oppliger, Rolf. *SSL and TLS : Theory and Practice.* 1st ed., Artech House, 2009.
- Bassett, L., Foley, M., Brown, K., Kwityn, J., Roumelioties, C., Troutman, E., Futato, D., Montgomery, K., & Demarest, R. (2015). *Introduction to JavaScript object notation : a to-the-point guide to JSON* (First edition.). O'Reilly.
- MITH, B. (2015). *Beginning JSON* (1st ed. 2015.). Apress.
- Hillar, G. C. (2017). *MQTT essentials - a lightweight IoT protocol : the preferred IoT publish-subscribe lightweight messaging protocol* . Packt Publishing.
- Chen, W.-J., Lampkin, V., & Gupta, R. (2014). *Responsive mobile user experience using MQTT and IBM MessageSight* (1st ed.). IBM Corporation International Technical Support Organization.
- Microsoft (2018). *Patrón de publicador y suscriptor.* Microsoft Docs.
- Pierdicca, Roberto & Marques-Pita, Manuel & Paolanti, Marina & Malinverni, Eva. (2019). *IoT and Engagement in the Ubiquitous Museum. Sensors*. 19. 1387..
- Piccialli, Francesco & Chianese, Angelo. (2014). Designing a Smart Museum: When Cultural Heritage Joins IoT.
- Minsait (2019), *Bienvenido al museo del futuro,* El confidencial.
- World Travel & Tourism Council. (2019). *Economic Impact Reports*.

- World Tourismo Organization. (2019). *Spain: Country-specific: Arrivals of non-resident tourists at national borders, by country of residence 2014 – 2018.*
- World Tourism Organization. (2019). *Spain: Country-specific: Basic indicators (Compendium) 2014 – 2018.*
- World Travel & Tourism Council. (2019). *SPAIN 2020 ANNUAL RESEARCH: KEY HIGHLIGHTS*.
- Cesar Dachary, A. A., & Arnaiz Burne, S. M. (2013). *El turismo y la sociedad de consumo. Anuario turismo y sociedad*, 14, p. 65.
- International Data Corporation. (2020). *Smartphone Market Share.*
- Schildt, H., & Coward, D. (2014). *Java* (6th edition). McGraw Hill Osborne.
- Oracle. (Sin Fecha). *The Java Language Environment.* Obtenido de: https://www.oracle.com/java/technologies/introduction-to-Java.html
- Gilbert, Stephen. (1997). *Object oriented programming in Java*. Pag. 16.
- Jaramillo Valbuena, S., Cardona Torres, S. A., & Hernández Rodríguez, L. A. (2010). *Programación orientada a objetos* . Ediciones Elizcom.
- Kreibich, J. A. (2010). *Using SQLite* (1st ed.). O'Reilly.
- *Royal Navy.* (2019). *The Admiralty Manual of Navigation Vol 1: Principles of Navigation* (11th edition).
- *The Stationery Office, 1987, p. 10*
- Owntracks (2021). Regions (Waypoints).
- Robledo Sacristán, C., & Robledo Fernández, D. (2012). *Programación en Android*. Ministerio de Educación, Cultura y Deporte.
- Komatineni, S., & MacLean, D. (2012). *Pro Android 4 [electronic resource]* (1st ed. 2012.). Apress.
- Perry, J. S. (2009). Log4J (1st edition). O'Reilly Media.
- Hebuterne, S. (2016). *Android: guía de desarrollo de aplicaciones Java para Smartphones y tabletas* (3ª ed.). ENI.