UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"PROTOTIPO DE SISTEMA DE SEGUIMIENTO GPS PARA PEQUEÑAS EMBARCACIONES DE VELA BASADO EN ANDROID"

TRABAJO FIN DE GRADO

Febrero – 2019

AUTOR: Saúl Fernández Candela

DIRECTOR/ES: Miguel Onofre Martínez Rach

Alejandro Pomares Padilla

1 INTRODUCCIÓN

1.1 Motivación

La idea que sustenta el presente trabajo parte de la intención de desarrollar un sistema de seguimiento y control de pequeñas embarcaciones de vela. Estas embarcaciones se encontrarían en un entorno controlado, ya sea en el ámbito de una competición de regatas o en el aprendizaje de tripulantes de corta edad. El sistema cumpliría la función de informar tanto a los monitores o jueces como a los propios padres de los pupilos de la situación de cada una de las naves y sus pilotos. En el mundo actual prácticamente todos tenemos al alcance de nuestra mano un *smartphone*, dispositivo que nos brinda múltiples maneras de comunicación, acceso a internet, geolocalización y capacidad de capturar imágenes. Teniendo en cuenta esto, la solución natural parece clara: crear una *app* capaz de recoger la posición GPS e imágenes de la embarcación, para mandarlas a un servidor, desde el cual estos datos puedan ser recogidos y leídos por el cliente.

1.2 Objetivos

Una vez expuesta la idea sobre la que se va a trabajar, el siguiente paso es definir metas concretas para poder llegar a una solución práctica. Las funciones básicas empezarían por la captación de la posición, utilizando el sensor GPS del dispositivo, junto a otros datos relevantes como el rumbo, la velocidad, y el tiempo a razón de una vez por segundo. La imagen capturada de la embarcación, se haría a razón de una cada 5 segundos, ya que significa una carga mayor de información para el servidor en comparación con los datos anteriores. Puesto que estas funciones están pensadas para funcionar durante todo el periodo de la regata, deberán ejecutarse en segundo plano, con el dispositivo en estado de suspensión. A continuación, los datos que se mandan deben ser almacenados en la base de datos, con el objetivo de ser leídos más tarde, por lo tanto será necesaria una API (Application Program Interface) que actúe como puerto de entrada y salida de información. Finalmente, es necesaria una interfaz en la que el cliente reciba la información. Esta interfaz implementará distintas funciones: seguimiento en directo de la posición, seguimiento en directo de la imagen, recorrido completo de la embarcación y marcador de inicio y final de las regatas, a modo de juez.

2 SOLUCIÓN PROPUESTA

2.1 Descripción y esquema de funcionamiento

Frente a los objetivos detallados, la opción que nos pareció óptima para ser desarrollada se estructura en tres pilares básicos, dos apps y un servidor. Las dos aplicaciones Android cumplirán los dos extremos del proceso; el inicial, con la recogida y envío de datos: la aplicación GPS Volley Location, y el final con la presentación de los datos de forma sencilla y atractiva al cliente: la aplicación App Regatas. Por otro lado, como nexo de unión, tenemos el servidor. Este alojará la base de datos así como los servicios web encargados de procesar la información de entrada y salida.

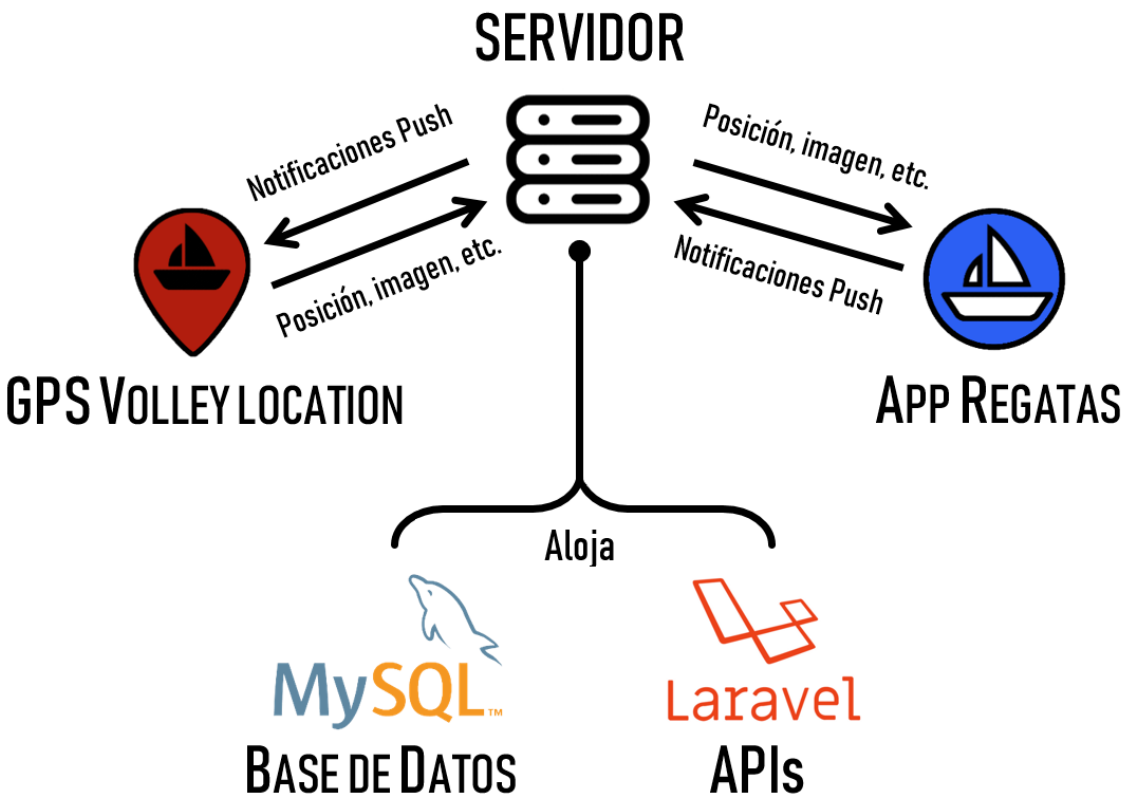

Figura 1: Esquema de funcionamiento del proyecto

La aplicación GPS Volley Location por lo tanto actuará como captador de información básicamente. Una vez puesta en marcha utilizará los sensores del dispositivo para registrar los datos y enviarlos al servidor. Requerirá una interacción mínima con el usuario, está pensada para ser consumida por el patrón de la embarcación de una forma pasiva e indirecta, razón por la cuál puede trabajar con la pantalla apagada.

App Regatas en cambio, es una aplicación centrada en proveer al usuario de la mimsa (padres, monitores, club, patrón, etc..) de distinta información, tanto en tiempo real como recreando a posteriori los movimientos e imágenes de la emabarcación en un periodo de tiempo. En función de qué se quiera visualizar, la app la obtendrá los datos del servidor, los procesará y presentará en la interfaz gráfica. Es decir, al contrario que GPS Volley Location, se consume de una forma activa.

Resumidamente, podríamos decir que GPS Volley Location sería el *backend* del proyecto, y App Regatas el *frontend*.

2.2 Límites del proyecto

La intención del proyecto es construir un **prototipo** que demuestre que la estructura planteada es funcional. Por este motivo el trabajo se encuentra acotado a la practicidad. La meta no es otra que satisfacer los objetivos planteados, lejos de ser un producto acabado o probado en un nicho de mercado, puesto que esto excede el marco de un trabajo final de grado.

Dicho esto, lo que se pretende en este apartado es contextualizar el escenario en el que se ha llevado a cabo el desarrollo del código de las aplicaciones, tanto Android como web. Esto no significa que el proyecto haya sido ejecutado de forma descuidada, o que los objetivos se hayan cumplido de forma inconexa como ítems de una lista de tareas. Simplemente se quiere hacer constancia de que es posible que surjan inconveniencias al poner el sistema a prueba en situaciones reales de estrés como pueden ser grandes competiciones. En estos casos el tráfico de información en el servidor es significante debido al gran número de embarcaciones, y el comportamiento de las aplicaciones y el servidor no ha sido estudiado ya que como avanzábamos antes, esto no formaría parte de los objetivos.

3 DESCRIPCIÓN DE LOS COMPONENTES Y TECNOLOGÍAS

3.1 Android Studio

Creado mediante IntelliJ IDEA, Android Studio es el entorno de desarrollo integrado (IDE) oficial para la plataforma Android, el famoso sistema operativo para dispositivos móviles táctiles basado en el núcleo Linux.

Gracias a él podremos desarrollar aplicaciones con las más recientes versiones de Android. En nuestro caso trabajamos con la versión estable 3.0.1 de Android Studio, y las aplicaciones están diseñadas para trabajar con la API 26 (Android Oreo), aunque con el objetivo de aumentar la compatibilidad a un mayor segmento de dispositivos, están preparadas para asegurar su funcionamiento hasta la API 21 (Android Lollipop).

Android Studio se apoya en tres lenguajes de programación, Java, Kotlin y XML. Los dos primeros no se complementan el uno con el otro, debemos elegir entre uno de ellos ya que se encargan de lo mismo, desarrollar la funcionalidad de la app, es decir, lo que queremos que "haga". Kotlin es un lenguaje orientado a objetos de reciente aparición que busca ser mejor que Java usando su biblioteca de clases. Aún así, en el proyecto se decidió utilizar Java porque es un lenguaje mucho más arraigado y que ya conocíamos. El XML por otro lado, es necesario para dar formato a la interfaz que va a controlar la app y permite al usuario interactuar con ella.

Como resultado, Android Studio ofrece un entorno para el desarrollo de aplicaciones muy potente y accesible para el usuario. Su interfaz permite desarrollar el diseño de la propia app mediante la simple mecánica de "arrastrar y soltar" los elementos o vistas (*views*) en el sitio que queramos situarlo dentro de la pantalla de ejecución de la app. De forma simultánea, si cambiamos de pestaña podemos ver que todo lo que hemos hecho de esta forma se ve reflejado en código XML, que también podemos escribir de forma directa si estamos más familiarizados con él. Por otro lado, el código Java no es tan sencillo en comparación ya que no existe esta mecánica de arrastrar y soltar. Aún así, disponemos de ayudas como autocompletar cuando llamamos a un método, resaltar errores cuando el tipo de datos no concuerda o hacemos referencia a un objeto que el compilador no reconoce.

Además, Android cuenta con una documentación completa en inglés, parcialmente traducida a español, de cada una de las clases y métodos que implementa y podemos encontrar en https://developer.android.com/docs.

3.2 Escritorio Remoto

El servidor donde se ha realizado el desarrollo con Android Studio, que dispone de mayores recursos hardware y por tanto velocidad de funcionamiento que el portatil del alumno, ha facilitado en gran manera el uso ágil de Android Studio. Este servidor se encuentra en los despachos del Departamento de Ingeniería de Computadores en el edificio Alcudia de la UMH.

Mencionar que la totalidad de las aplicaciones han sido creadas, probadas y depuradas utilizando la tecnología de Escritorio remoto de Windows conectándose al servidor de desarrollo. El escritorio remoto nos permite controlar en una terminal la interfaz gráfica (escritorio) de un ordenador que se encuentra en otro lugar. Desgraciadamente, el efecto colateral del uso del Escritorio remoto de Windows, es que éste impide la depuración en un dispositivo físico mediante USB. Para solucionar este aspecto se vinculó a la máquina remota los dispositivos de depuración, y gracias a una aplicación Shell Script (vincular-usb) desarrollada por el Departamento y de un plugin de Android Studio () se pudo realizar una depuración remota sin cables en los dispositivos móviles vinculados, aunque éstos no podían apagarse para mantener el enlace. Esta aplicacción hace uso del ADB (Android Debug Bridge), funcionalidad que permite vincular un dispositivo mediante su IP una vez conectado a la red del equipo anfitrion. Los dispositivos móviles vinculados se conectan mediante una VPN debidamente securizada al host donde reciben una IP y se configuran para permitir el acceso a los puertos USB conectados remotamente.

3.3 Servidor

Como es natural, para desarrollar un proyecto que utilice servicios en línea como el nuestro (base de datos, API), es necesario un servidor que los aloje. La opción elegida es la distribución Debian de Linux, el software libre ampliamente conocido, concretamente la versión Debian 9.3, conocida como *Stretch*. Debian se caracteriza por su estabilidad, cuenta con más de 20 años lanzando versiones estables, testeadas en un vasto público

gracias a su libre distribución. Por esto, consideramos que es la mejor opción para nuestro servidor.

Ahora, para desempeñar tanto el servicio web como el de la base de datos, es necesario contar con otros componentes en nuestro servidor. Para llevarlo a cabo se ha optado por utilizar el paquete XAMPP, el cual incluye Apache, MariaDB, PHP y Perl. Utilizaremos los tres primeros. Apache es un software de código abierto con el que funcionan casi la mitad de los servidores web en el mundo. Con él podremos levantar el punto de acceso al que acudirán las aplicaciones para enviar sus datos. MariaDB es la versión con licencia pública de MySQL. Nos servirá para almacenar los datos recibidos en nuestro servicio web. Hablaremos más de este componente en puntos siguientes. Por último, PHP es un lenguaje interpretado y de propósito general especialmente adecuado para el desarrollo web. Es de código abierto y se suele describir como lenguaje de lado del servidor, lo que significa que ante una entrada proveniente de un usuario, el código se ejecuta en el propio servidor y se envía la salida al usuario, sin saber éste último el contenido del script.

Debian no solo es un servidor, es un proyecto respaldado por una comunidad de desarrolladores, usuarios e incluso cuenta con el apoyo de varias empresas en forma de infraestructuras. Así, consta con una página web (https://www.debian.org) donde podemos encontrar todo acerca de Debian, incluido un detallado manual con su proceso de instalación y una sección de soporte llena de ayudas, por lo que nos referimos a ella como el mejor recurso para instalar este componente.

3.3.1 XAMPP

Pese a que este paquete no posee unas dimensiones como las de Debian, también es un recurso ampliamente conocido y utilizado por la comunidad de desarrolladores, por lo que la red está llena de manuales, tutoriales, y consultas resueltas sobre la puesta a punto de la herramienta. Por este motivo, también instamos a recurrir a ella para su instalación y configuración en el servidor Debian.

3.3.2 MySQL y MySQL Workbench

Probablemente el sistema de gestión de bases de datos relacional más conocido y usado, con clientes como YouTube, PayPal, Google o Facebook. ¿Pero qué es una base de datos relacional? ¿Y qué hace a MySQL tan buen sistema de administración?

En todos los ejemplos de clientes que hemos citado, podemos ver con claridad que detrás de sus servicios hay millones y millones de datos que de una manera u otra almacenan e indexan. Evidentemente, esa información no se guarda en una única y gran tabla, sino que se separa en tablas más pequeñas que guardan relación entre sí con el fin de obtener una mayor velocidad y flexibilidad. Cuanta más información se trata, mayor es el número de tablas y relaciones de dependencia entre ellas, y aquí es donde aparece la utilidad de los sistemas de administración en las bases de datos. Gracias a ellos podemos acceder, analizar y modificar tanto datos como tablas de forma consistente mediante *lenguaje de consulta estructurada*, mejor conocido por sus siglas en inglés, SQL.

Desarrollado en ANSI C y C++, MySQL constituye uno de los proyectos de código abierto más grandes del mundo, pese a que actualmente posee una licencia dual, la Licencia Pública General (GNU GPL) y la Licencia comercial, por Oracle Corporation. Esto significa que cualquier persona puede acceder al código fuente de MySQL, modificarlo a su gusto y utilizarlo para fines propios de forma gratuita, siempre que se ajuste a la GNU GPL. Pero por otro lado, si se pretende explotar sus capacidades en aplicaciones comerciales, es necesario pagar una licencia comercial. Así, como nos encontramos en un ámbito educacional de desarrollo y prototipado del proyecto, utilizamos la derivación GPL de MySQL: MariaDB.

Para trabajar el esquema relacional que gobierna las tablas de la aplicación utilizaremos el programa MySQL Workbench, un entorno visual de diseño de bases de datos. Gracias a él podremos definir de forma sencilla las correlaciones entre las tablas, la estructura de cada una de ellas y el tipo de datos que alojan.

La estructura de la base de datos se construye sobre los elementos que conforman una regata. Se resume en el siguiente esquema:

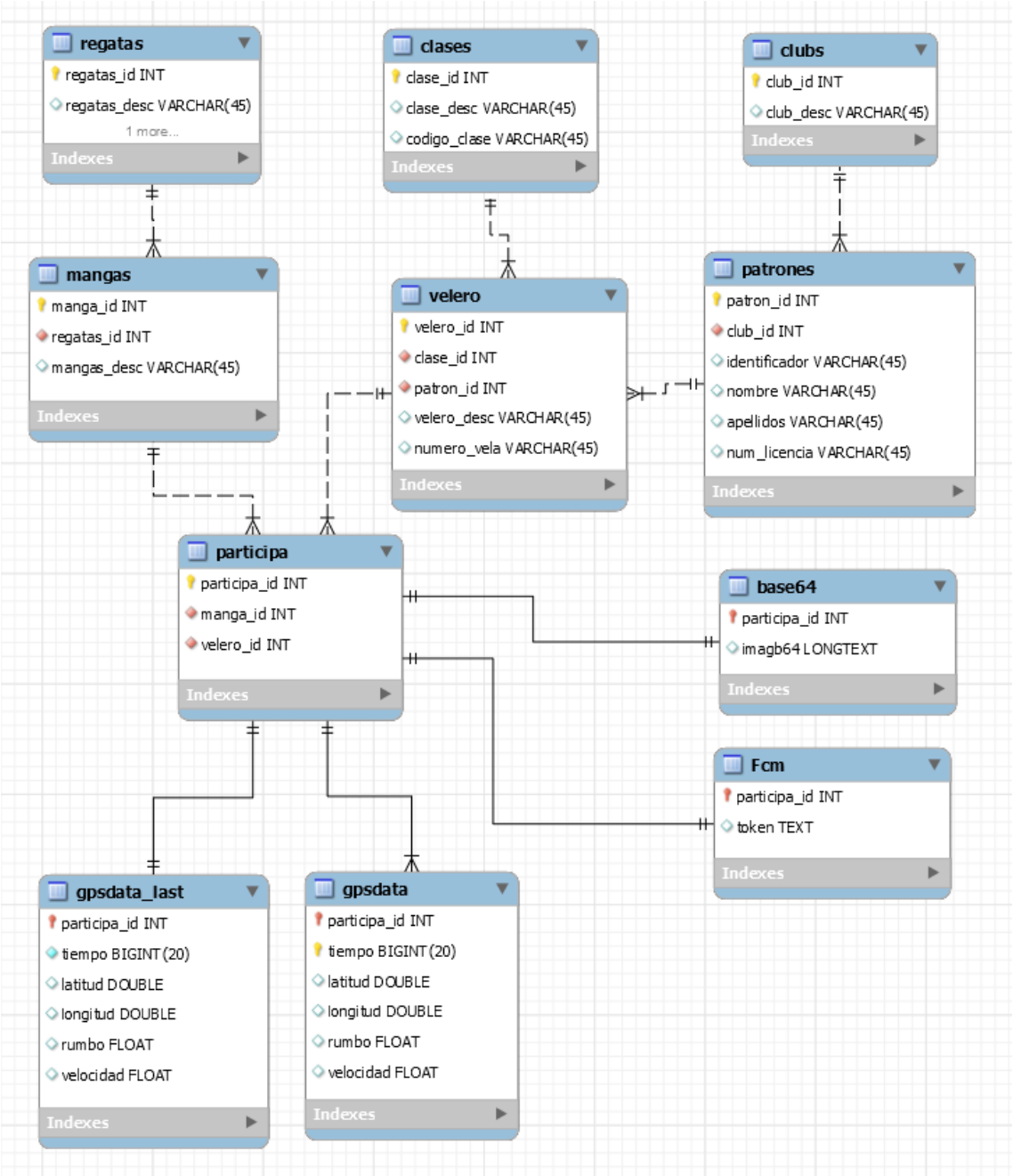

Figura 2: Estructura de la base de datos

Como podemos observar, es un esquema que incluye bastantes tablas, con muchos detalles como los tipos de veleros, nombre e identificación de los patrones, club al que pertenecen, etc. Esto nos da una buena visión de la forma que adoptaría el proyecto al llegar a un producto acabado además de proporcionar una vista general de cómo funcionan las regatas. Sin embargo, en este proyecto utilizaremos un conjunto reducido de tablas, centrándonos en el uso que les darán las aplicaciones y la API para cumplir los objetivos del proyecto, estas son:

- **Participa:**

  La tabla recoge la participación de un velero en una manga de una regata. Una regata es un evento en el que tienen diversas competiciones o mangas. En estas mangas es donde compiten los veleros, haciendo el recorrido que nosotros tenemos que registrar. Por lo tanto, el campo *participa_id* es el registro que utilizaremos para identificar cada participación diferente de un velero en una manga.

- **Gpsdata:**

  Tabla en la que se almacenarán todas las posiciones GPS de las participaciones de inicio a final de la manga. Cada registro de esta tabla contiene los campos de tiempo, latitud, longitud, rumbo y velocidad.

- **Gpsdata_last:**

  Encargada de registrar solo la última posición GPS recibida. Cuando desde la aplicación de seguimiento se pida la posición actual del velero en un proceso de seguimiento se realizarán multiples y continuas consultas a esta tabla a intervalos fijos de tiempo. Esto, cuando se realice para todos los veleros participantes en una manga, puede suponer una carga elevada al gestor de bases de datos. Por tanto, al contener la tabla un único registro la consulta será muy rápida y facilitará dichas consultas masivas.

- **Base64:**

  En esta tabla se alojará la última imagen captada de la embarcación. Al almacenar únicamente la ultima captura, sólo cuando se realice un seguimiento en tiempo real de las imágenes del velero podremos tener una simulación de vídeo en directo (a frame rate reducido) de lo que ocurre en la embarcación. Al no guardarse todas las capturas no podrá reproducirse el vídeo en una sesión posterior. Esto se puede modificar pero en principio se ha configurado así por reducir el volumen de datos que supondría el guardar todas las capturas (de todas los veleros en todas las mangas de todas las regatas).

- **Fcm:**

  Tabla necesaria para almacenar los tokens de las aplicaciones. Se hablará de ellos más adelante.

3.4 Laravel

Se trata de un entorno de trabajo de PHP, su forma de entender el desarrollo de código. Laravel plantea una arquitectura de desarrollo web propia pero a su vez influenciada por otros *frameworks* que toma de partida el patrón MVC (Modelo, Vista, Controlador). Este patrón se fundamenta en tres componentes, los cuales le dan el nombre a la arquitectura de software. El primero de ellos, el Modelo, representa la información y formato con el que trabaja el sistema. Contiene la lógica de negocio. El siguiente es la Vista y su función es visualizar los datos contenidos por el modelo. Constituye una de las partes más importantes de la interfaz de usuario (IU), transmitir al cliente la información como salida del sistema. Finalmente, el Controlador es el encargado de otra parte fundamental de la IU que es responder a los eventos, normalmente propiciados por el usuario. El resultado del evento desemboca en una manipulación del modelo (actualización, consulta, borrado, etc. de un registro en la base de datos) o de la vista (scroll, pasar a ver más opciones en la siguiente página).

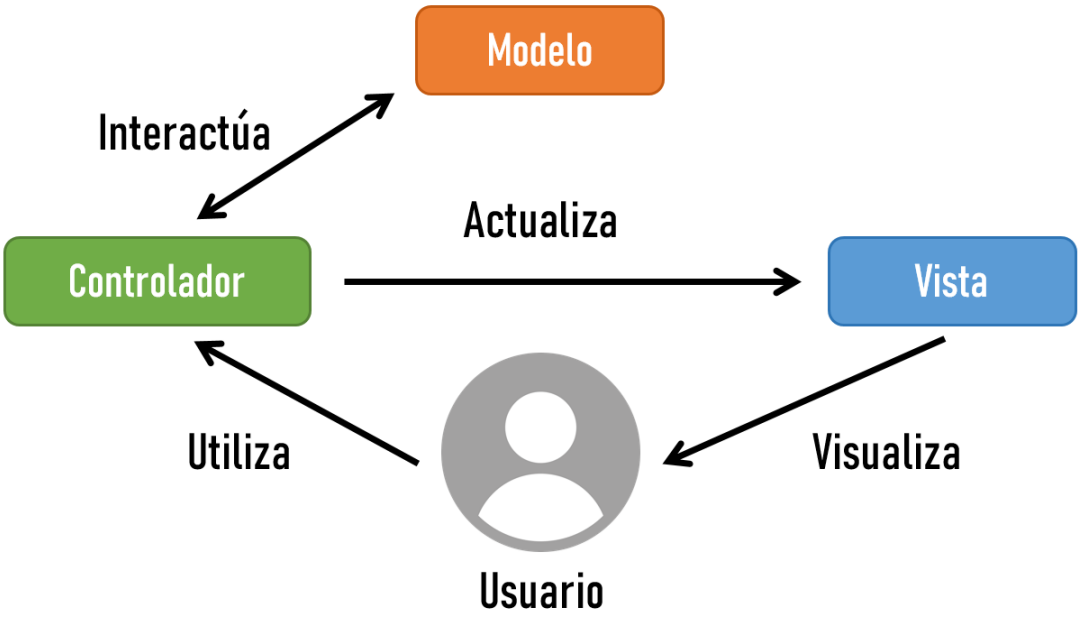

Figura 3: Esquema MVC

Este esquema se traslada a Laravel en los modelos, *blades* (motor propio de plantillas para las vistas) y controladores. Sin embargo, no se queda ahí. Laravel posee un sistema de ruteo que une los *blades* con las URLs y sus controladores de forma sencilla. También cuenta con mapeo objeto-relacional u ORM (Object-Relational Mapping) que recibe el nombre de Eloquent, fácil de usar y muy potente. Gracias a él nos ahorraremos líneas y

líneas de código al desarrollar nuestras funciones, ya que establece una relación en memoria (objetos) de los elementos de la base de datos relacional.

Todas estas herramientas que caracterizan a Laravel suscriben una filosofía *zen* que busca dar a los proyectos una organización limpia, fácil de leer y elegante. Y es que Laravel se presenta como "El entorno de trabajo PHP para artesanos". De hecho lo primero que podemos leer si accedemos a su página web es "¿Te gusta que el código sea bonito? A nosotros también". Este estilo se lleva a cabo incluso en el diseño y estructura de la propia página de documentación de Laravel (https://laravel.com/docs), donde podemos encontrar desde explicaciones sencillas sobre como instalarlo, hasta como utilizar Eloquent, pasando por temas de seguridad y autentificación.

Por las presentes razones, utilizaremos este framework para escribir el código de las APIs que harán posible la comunicación App → Servidor y Servidor → App.

3.4.1 Instalación Laravel

Laravel utiliza Composer, un administrador de dependencias para PHP. Su función básicamente es descargar e instalar todos los archivos necesarios para que funcione Laravel y sus extensiones. Composer funciona mediante la ejecución de líneas de comando en el símbolo del sistema, característica que se extiende a Laravel, y que utilizaremos para crear el proyecto. Para información más detallada sobre el proceso de instalación, consultar Anexo I.

3.4.2 Entorno

Los directorios y estructura de Laravel permiten realizar un amplio abanico de páginas web y APIs, donde todos los directorios y archivos tienen su función. Sin embargo, explicar el funcionamiento de cada uno se saldría fuera de los objetivos del proyecto, por lo que nos centraremos en las partes utilizadas en este proyecto: rutas, controladores y modelos.

3.4.2.1 Rutas

Este directorio está ubicado en la carpeta raíz del proyecto (*routes*) y en él se definen todas las rutas de la aplicación. La función que cumplen las rutas, es bastante obvio: ofrecen una URL por la que acceden las peticiones HTTP y les asigna un controlador.

Básicamente una ruta es una puerta que une una dirección HTTP con una función. Son llamadas de forma externa, y dependiendo del tipo de ruta que sea recibirá o dará información, y ejecutará la función que tenga asignada.

Por defecto, encontramos los siguientes archivos: *api.php*, *web.php*, *console.php*, y *channels.php*. Cada uno contiene unas características específicas y pertenece a un grupo de *middleware* (software que actúa como puente entre las peticiones HTTP y las respuestas). Centrándonos en el archivo que utilizaremos, *api.php*, está pensado para alojar rutas sin estado, base de las API RESTful. Todas las URLs instanciadas en este fichero, poseen el prefijo "*/api*".

```
Route::get('pos', 'GpsData@getAll');
```

En el ejemplo podemos identificar tres parámetros que conforman la ruta:

- **Tipo de petición:**

  En el ejemplo corresponde a la petición *get*, pero Laravel además soporta los métodos *post*, *put*, *patch*, *delete* y *options*.

- **URI:**

  Completa la dirección URL. Representado por una cadena de caracteres, *'pos'*, en la muestra. El resultado final sería "*/api/pos"*.

- **Función del controlador:**

  El último parámetro de la ruta. Hace explícito la función del controlador que llevará a cabo la ruta, donde *GpsData* corresponde al controlador y *getAll* el método específico que se llamará.

3.4.2.2 Controladores

Los controladores son básicamente *scripts* que se llevan a cabo la lógica que hay detrás de una petición. Estas peticiones llegan al sistema a través de una ruta. Al igual que una clase, los controladores, recogen diversas funciones que normalmente van enfocadas a trabajar con un mismo objeto o concepto. Representan la parte más sustancial desde el punto de vista de las APIs, por lo que desarrollaremos más su uso en el proyecto en su correspondiente sección. Podemos generar archivos de este tipo mediante el comando:

```
php artisan make:controller Controlador
```

Encontraremos los controladores bajo el directorio *app/Http/Controllers* de la instalación de Laravel

3.4.2.3 Modelos

Al igual que los controladores, se trata de un término directamente heredado del patrón de diseño MVC. Los modelos constituyen la relación entre el objeto con el que se trabaja en el código PHP y el registro de una tabla de una base de datos. Es el vehículo que se utiliza para interactuar con dicha tabla. Permiten tanto consultar, modificar einsertar nuevos registros.

Laravel está pensado para crear los modelos vía línea de comando:

```
php artisan make:model Modelo
```

Esto formará un archivo PHP con el esqueleto por defecto de la estructura del modelo, y que se situa por defecto en la ubicación *app/Http*. Sin embargo, debido a que hemos diseñado la estructura de la base de datos mediante MySQL Workbench y las tablas ya se encuentran creadas, la solución más sencilla para definir los modelos es importarlos desde la base de datos a nuestro proyecto de Laravel mediante el plugin Reliese.

Este plugin se conecta a nuestra base de datos cuya conectividad está definida en el archivo *.env*. Una vez conectado a la base de datos pasa a leerla y generar de forma automática todos los modelos, donde se crea un modelo por cada tabla. Así se evita tener que crear cada modelo de forma manual vía comando artisan, en el que tenemos que escribir en cada archivo las propiedades de cada uno. Para más información sobre el plugin Reliese, consultar el Anexo II.

3.4.3 API

Una API, si no estamos familizarizados con el término, es un conjunto de funciones y métodos que ofrece cierto software para poder ser utilizado por otra tecnología diferente y habitualmente de forma remota. Poniendo nuestro proyecto como ejemplo, la API que desarrollamos es el puente que hace posible que dos tecnologías distintas con lenguajes distintos como Android y Debian/XAMPP puedan trabajar juntas.

En este apartado describiremos las funciones que han sido desarrolladas para hacer posible la comunicación App → Servidor y Servidor → App. Se explicarán a grandes rasgos el funcionamiento de cada una, para más detalles sobre su implementación, se recomienda leer el Anexo III.

El funcionamiento de las APIs se basa en el uso de las peticiones HTTP, o mejor dicho en este caso, en la respuesta a éstas. Las peticiones HTTP son mensajes con una estructura predefinida: línea inicial, cabecera y cuerpo del mensaje (este último, es opcional). Estos mensajes son enviados al servidor para acceder o modificar un recurso y cuando el servidor lo recibe, devuelve una respuesta que puede ser vacía, ser la notificación de un error o la confirmación de que ha tenido éxito.

En nuestro proyecto de Laravel, todas las funciones que controlan la API se encuentran agrupadas en un mismo controlador, *GpsData.php*. En esta documentación, las estructuraremos según el tipo de petición HTTP: *GET* y *POST*. Existen muchos otros tipos, cada uno con sus características y usos específicos pero estos dos son los más comunes y por lo tanto, los utilizados en el proyecto.

Todas las funciones desarrolladas en el proyecto están preparadas para recibir y devolver datos en JSON, un formato de texto ligero de intercambio de datos estructurado en pares nombre/valor. Es especialmente útil porque tanto leerlo como escribirlo resulta simple para humanos y máquinas.

3.4.3.1 Peticiones POST

El método POST hace uso del cuerpo del mensaje para enviar información al servidor con la que crear o actualizar un recurso.

- **Función newPosition:**

  Será la utilizada cada vez que se quiera guardar una nueva posición en la base de datos. La función está diseñada para recibir toda la información relativa a la embarcación excepto la imagen: identificación de participación, tiempo, latitud, longitud, rumbo y velocidad. Aprovecha la misma petición para guardar la posición tanto en la tabla que registra el recorrido completo como en la tabla dedicada a almacenar la última posición.

- **Función postImg:**

  Función preparada para guardar la última imagen recibida, junto a su correspondiente identificación de participación.

  Las siguientes dos funciones utilizan métodos del plugin Laravel-FCM, responsable junto a la API Firebase Cloud Messaging (FCM) de Google, de que las notificaciones push sean posibles. Para más información sobre esto, consultar el Anexo II y III.

- **Función postToken:**

  Las notificaciones push están diseñadas para poder enviar mensajes tanto a un dispositivo en concreto, como a un conjunto de los usuarios suscritos a un tema específico, o a la totalidad de ellos. Por este motivo es necesario que la primera vez que un cliente utiliza la aplicación, mande a la base de datos una clave identificadora única llamada *token*. Esta función se encarga de recibir el *token* y almacenarlo.

- **Función sendFcm:**

  Se encarga de enviar la notificación push con la ayuda de la API FCM de Google a todos los usuarios con el objetivo de informar qué regata va a empezar o terminar. La función recibe el mensaje con la identificación de la regata y una clave binaria que indica el comienzo o finalización, construye la notificación con los metadatos anteriores como cuerpo, y se manda a la aplicación. Posteriormente la app recibidora de la notificación utilizará esos datos para habilitar/deshabilitar el envío de información a la base de datos.

3.4.3.2 Peticiones GET

El método más utilizado en el protocolo HTTP. Se utiliza para solicitar información de un recurso en concreto.

- **Función getAll:**

  Devuelve todas las posiciones GPS registradas a lo largo del tiempo en forma de un array JSON.

- **Función getLast:**

  Facilita la última posición GPS guardada. Contiene la latitud, longitud, tiempo, rumbo y velocidad asociada a la identificación de la participación.

- **Función getImg:**

  Proporciona la última imagen almacenada en la base de datos.

3.5 Postman

Programa que utilizaremos para probar la API, ya que al tratarse de código PHP, éste necesita recibir peticiones externas para comprobar su funcionamiento. Postman permite realizar peticiones mediante una interfaz intuitiva y fácil de usar en la que tan solo necesitaremos que ingresar la URL a la que responde la API, el tipo de petición HTTP que queremos hacer (GET, POST, etc.) y el cuerpo con los datos que transporta, si es necesario. Podríamos decir por lo tanto que Postman actuará como herramienta de depuración del servidor, ya que gracias ella podremos probar el funcionamiento de la API y la base de datos, para así poder arreglar los posibles errores que presenten.

En su página web (https://laravel.com/docs), donde podemos descargar de manera gratuita la aplicación de escritorio, informarnos detalladamente sobre todas sus utilidades y acceder a la documentación.

4 APLICACIONES ANDROID: COMPONENTES BÁSICOS

Para una mayor profundidad, se sugiere consultar los anexos que se citan y bibliografía/webgrafía. En esta última se encuentran todas las referencias que se han tenido en cuenta para desarrollar el proyecto, clasificadas por temática.

La capa de Android del proyecto abarca dos aplicaciones: GPS Volley Location y App Regatas. Cada una tiene funciones distintas y están pensadas para ser utilizadas por clientes diferentes, sin embargo, poseen, como cualquier aplicación Android, varios elementos comunes básicos para su funcionamiento, que explicaremos antes de entrar en detalle en cada aplicación. Para más información acerca de los componentes básicos a la hora de desarrollar una aplicación Android, se recomienda visitar https://developer.android.com/training/basics/firstapp.

4.1 Activity

Unidad básica de funcionamiento de las apps. Representan la pantalla con la que los usuarios pueden interactuar, desde marcar un número de teléfono, tomar una foto o enviar un correo electrónico hasta ver un mapa. Cada actividad tiene asignada una ventana en la que se puede dibujar su IU, el llamado *layout*, archivo escrito en XML que define los elementos que componen la interfaz gráfica. Están diseñadas para desempeñar una función en concreto o varias que tengan relación entre ellas. Normalmente, una de las actividades se califica como principal, la *Main Activity*, y es la que se lanza como bienvenida cuando se abre la app por primera vez. A partir de ésta se pueden lanzar otras actividades con funcionalidades distintas, perdiendo el foco de atención (lo que se está visualizando en la pantalla del dispositivo) la actividad principal para dárselo a la nueva. Esto no significa que se destruya la actividad principal con sus datos e interfaz, solo que pasa a un estado distinto. Cada uno de estos estados son representados por un método de la clase y a su vez todos estos métodos constituyen el ciclo de vida de las actividades.Dentro de este ciclo, podemos hacer tres distinciones:

- **Ciclo de vida completo:**

  El ciclo más extenso y que por lo tanto debe cumplir las funciones más generales de la actividad. Empieza con el método *onCreate*, donde se debe configurar su estado global y diseño, y termina con el método *onDestroy*, pensado para liberar todos los recursos restantes.

- **Ciclo de vida visible:**

  Acotado por los métodos *onStart* y *onStop*, este ciclo corresponde al tiempo de ejecución en el que la actividad es visible por el usuario y puede interactuar con ella. Dentro del ciclo se deben registrar y conservar los recursos necesarios para la IU básicamente.

- **Ciclo de vida en primer plano:**

  Transcurre entre las llamadas a los métodos *onResume* y *onPause*. La actividad se encuentra al frente, tiene el foco de atención del usuario para interactuar con él. Se entra y sale con frecuencia de este ciclo, cada vez que el dispositivo entre en suspensión o aparece un diálogo y se devuelve el foco a la actividad, por eso el código dentro de estos métodos debe ser ligero, para facilitar la fluidez de la app.

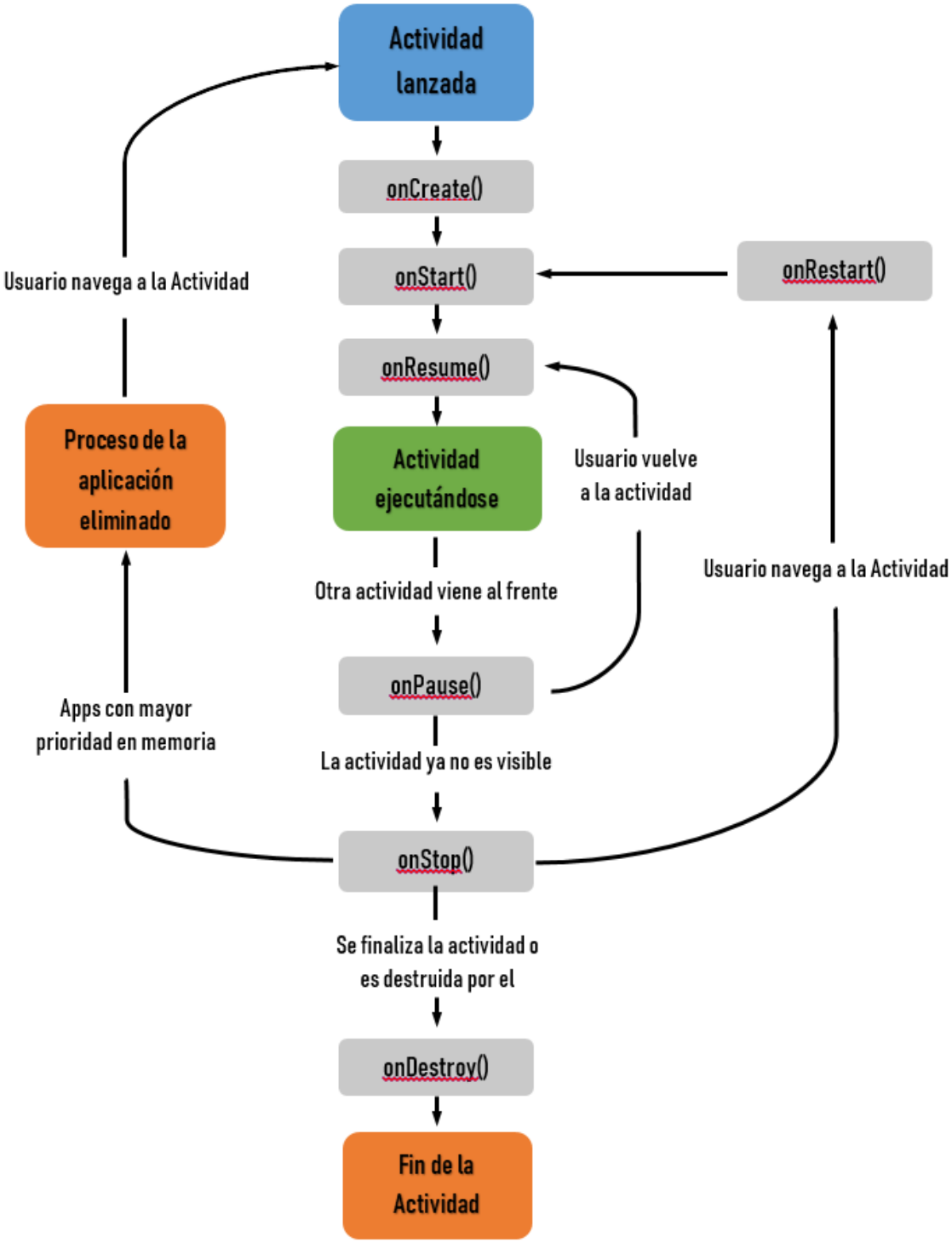

Figura 4: Ciclo de vida de una actividad

4.2 Layout

Traducidos al español como “diseño”, son la estructura visual que Android utiliza como IU para las actividades. Como se mencionó anteriormente, pueden ser creados mediante la declaración de sentencias en XML en el propio editor de texto de Android Studio o bien mediante la creación de una instancia de elementos del diseño en tiempo de ejecución.

La ventaja que supone declarar la IU en XML es que permite separar de una forma más clara y organizada la presentación de la aplicación del código que la controla. Así, puedes modificar la interfaz sin tener que modificar también el código fuente y volver a compilarlo.

4.3 Volley

Se trata de una librería HTTP cuyo objetivo es hacer más fácil y rápido las conexiones de Android con la red. Fue introducida en el evento Google I/O de 2013 por Ficus Kirkpatrick y como él mismo explica en la presentación, “cosas como acceder a APIs JSON o cargar imágenes son triviales para Volley”. Por lo tanto, Volley significa una herramienta de gran utilidad, ya que trabajar con APIs JSON y cargar imágenes es justo lo que necesitamos. Además, Volley cuenta con una programación automática de las peticiones que nos permite olvidarnos de escribir código de forma repetitiva, cosa que ocurriría si utilizáramos la opción alternativa, las *AsyncTasks*.

Para explicar el funcionamiento de Volley, ya que se trata seguramente de la parte más esencial de las apps, utilizaremos un pequeño ejemplo de una petición HTTP.

```
final TextView mTextView = (TextView) findViewById(R.id.text);
RequestQueue queue = Volley.newRequestQueue(this);
String url ="http://www.google.com";
```

Primero instanciamos la vista en la que mostraremos el resultado de la petición, *mTextView*. Acto seguido, la cola de peticiones *queue*, que será la encargada de utilizar y la cadena de caracteres *url* con la dirección web de la API a consultar.

```
StringRequest stringRequest = new StringRequest(Request.Method.GET,
url,
    new Response.Listener<String>() {
    @Override
    public void onResponse(String response) {
```

```
        // Muestra los primeros 500 caracteres de la respuesta
        mTextView.setText("La respuesta es : " +
response.substring(0,500));
    }
}, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {
        mTextView.setText("Algo ha ido mal!");
    }
});
```

Ahora creamos un objeto de tipo *stringRequest*, que recibe tres parámetros. El primero corresponde al tipo de petición, en este caso GET, el segundo la URL y en el tercero se sobrescribe el *listener* que controla la respuesta: *onResponse* para una petición exitosa, y *onErrorResponse* para una petición que encuentre algún tipo de error.

```
// Añadimos la petición a la cola
queue.add(stringRequest);
```

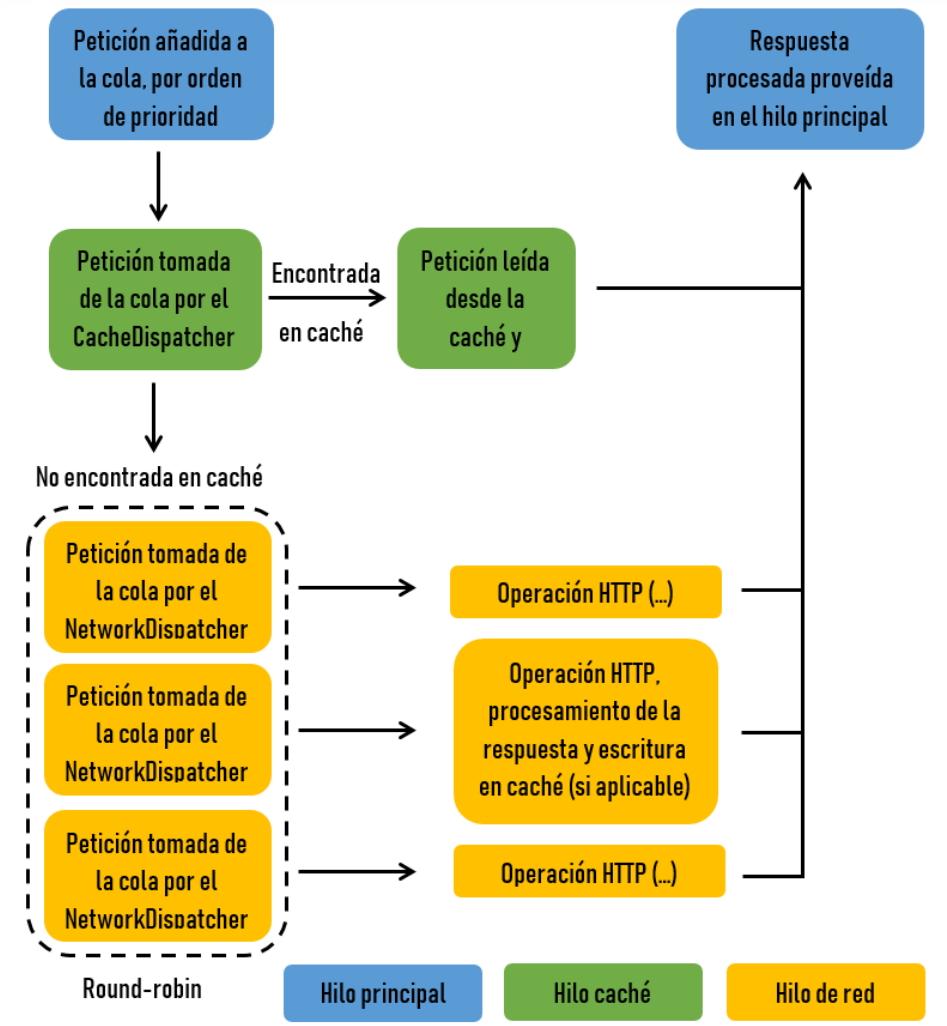

Figura 5: Esquema de funcionamiento de una petición en Volley

Finalmente se ejecutará la petición añadiéndola a la cola mediante el método *add* del objeto *queue*, correspondiente a la cola de petición.

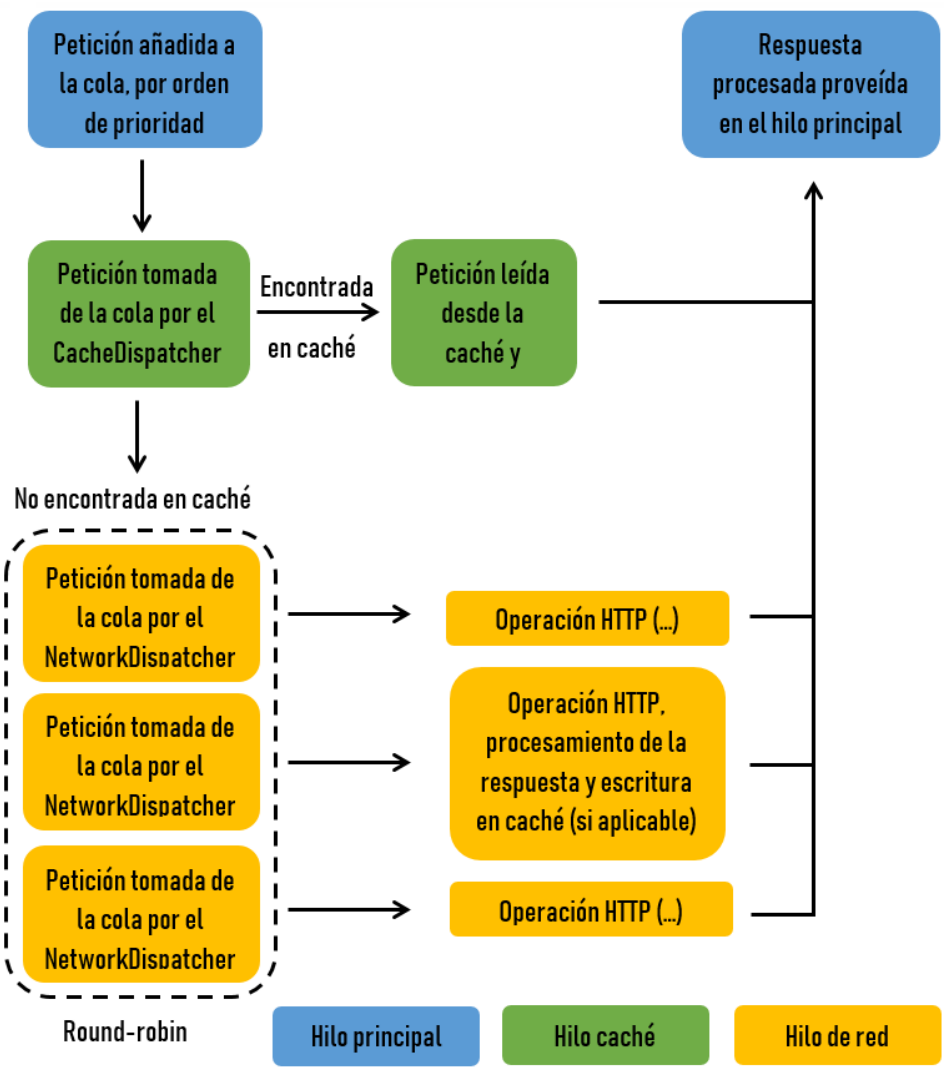

Figura 6: Esquema de funcionamiento de una petición en Volley

Bien, ahora que ya sabemos cómo utilizar Volley, ¿cómo funciona? Una vez se llama al método *add*, Volley pone a trabajar un hilo de ejecución en la caché y un conjunto de hilos de operaciones en red. La petición pasa por el hilo en caché y se evalúa: si la petición puede ser servida desde caché, se procesa la respuesta y se lleva el resultado al hilo principal. Si no es así, la petición se lleva a la cola de red. El primer hilo de operaciones en red que se encuentre disponible toma la petición y hace la operación HTTP. Se procesa la respuesta y se lleva de vuelta al hilo principal.

4.4 Interfaz

A diferencia de Volley, no se trata de una librería desarrollada por Google. Las interfaces, y en este caso **no nos referimos a las de usuario**, son una colección de métodos abstractos y propiedades constantes en Java. Nos serán de gran utilidad en ambas aplicaciones para implementar Volley en las diferentes partes que las constituyen sin necesidad de repetir el código en cada una de ellas.

Las interfaces no pueden ser instanciadas, solo implementadas por clases o extendidas por otras interfaces, y puesto que las implantaremos en ambas aplicaciones y sus distintas clases y servicios, explicaremos sin entrar en detalles cómo lo hemos llevado a cabo. Se puede encontrar más información en el Anexo IV.

- **Creación de la propia interfaz:**

  No tiene mayor complicación, se escriben los nombres y argumentos de las funciones abstractas que se quieren implementar.

- **Creación de una clase Singleton para Volley:**

  Como nuestra intención es simplificar el uso de Volley en las aplicaciones, la idea de evitar repetir código pasa por unificar todas las peticiones bajo una misma cola. Por esto se crea una clase *singleton* de Volley, porque permite restringir la creación de objetos de cierta clase. En este caso, como mencionábamos, de la cola de peticiones de Volley, *RequestQueue*. En resumen, esta clase asegura que solo haya una cola de peticiones en la aplicación.

- **Creación de una clase de ayuda:**

  Clase que contiene qué harán los métodos de Volley a implementar, ya que en la interfaz solo se especifican las llamadas a éstos, no el código en sí. Éste es el lugar del código que no queremos repetir en cada clase en la que queramos implementar Volley. Posee también el constructor o constructores con todos los parámetros necesarios para poder realizar las peticiones y distinguirlas, pues al tratarse de una interfaz, son métodos abstractos que pueden venir de cualquier parte del código de la aplicación.

4.5 Manifiesto

Archivo tipificado que toda aplicación Android debe tener. Se encuentra en el directorio raíz siempre bajo el mismo nombre, *AndroidManifest.xml*. Proporciona información esencial sobre la app al sistema operativo Android como:

- Paquete Java de la aplicación que funciona como identificador único de esta.
- Descripción de los componentes que forman la aplicación: actividades, servicios, receptores de mensajes y proveedores de contenido.
- Nombre de las clases que implementa cada uno de los componentes y sus capacidades que notifican al sistema Android los componentes y las condiciones para el lanzamiento.
- Determina los procesos que alojan a los componentes de la aplicación.
- Declara los permisos que debe tener la aplicación para acceder a las partes protegidas de una API e interactuar con otras aplicaciones así como los permisos que otros deben tener para interactuar con los componentes de la aplicación.
- Enumera las clases *Instrumentation* que proporcionan un perfil y otra información mientras la aplicación se ejecuta. Estas declaraciones están en el manifiesto solo mientras la aplicación se desarrolla y se quitan antes de la publicación de esta (No se han utilizado en nuestro desarrollo).
- Declara el nivel mínimo de Android API que requiere la aplicación.
- Enumera las bibliotecas con las que debe estar vinculada la aplicación.

4.6 Permisos

Android es un sistema operativo con privilegios independientes, en el que cada aplicación se ejecuta con una identidad de sistema distinta (ID de usuario de Linux e ID de grupo). También se separan partes del sistema en identidades distintas. Así, Linux aísla las aplicaciones entre sí y del sistema operativo. Por esto, Android ofrece funciones de seguridad adicionales mediante su mecanismo de permisos; restricciones aplicadas a operaciones específicas que dan acceso a datos que pueden comprometer la privacidad del usuario o funcionamiento del dispositivo.

4.6.1 Tipos de permisos

Existen tres tipos de permisos en Android, los permisos *normales*, los permisos *especiales* y los llamados permisos "*riesgosos*".

- **Permisos normales:**

  Abarcan áreas en las que la app tiene que acceder a datos o recursos fuera de su ámbito estricto de funcionamiento, pero solo existe un riesgo mínimo para para la privacidad del usuario, por los que si son requeridos por la app, el sistema se los otorga automáticamente. Un ejemplo de este tipo sería consultar el huso horario del dispositivo.

- **Permisos especiales:**

  Los permisos especiales tan solo implican dos permisos en sí, concretamente *SYSTEM_ALERT_WINDOW* y *WRITE_SETTINGS*. Son particularmente confidenciales y la mayoría de aplicaciones no deben usarlo. No se utilizan en el proyecto por lo que no haremos más hincapié en ellos.

- **Permisos riesgosos:**

  Requieren recursos que incluyen información privada del usuario o que podrían afectar el funcionamiento de otras apps. Funciones que necesitan dicho permiso son por ejemplo leer los contactos del usuario, los mensajes SMS que recibe o incluso enviarlos. Si la aplicación necesita los permisos, serán pedidos al usuario conforme esta se abra, y una vez dados no se volverán a pedir, a no ser que sean revocados.

5 APLICACIÓN GPS VOLLEY LOCATION

Aplicación encargada de captar y enviar los datos recogidos por el dispositivo, ubicado en la embarcación. Está diseñada para tener la mínima interacción con el cliente, principalmente para poder trabajar con la pantalla apagada y así reducir de forma significante el uso de batería. La única acción por parte del usuario que se requiere para que empiecen o paren los servicios es pulsar el botón *Start* o *Stop*, respectivamente.

5.1 Estructura

En este apartado se presenta la distribución interna de la aplicación. Mediante el siguiente esquema de la figura se pueden ver los archivos más importantes sobre los que se sustenta:

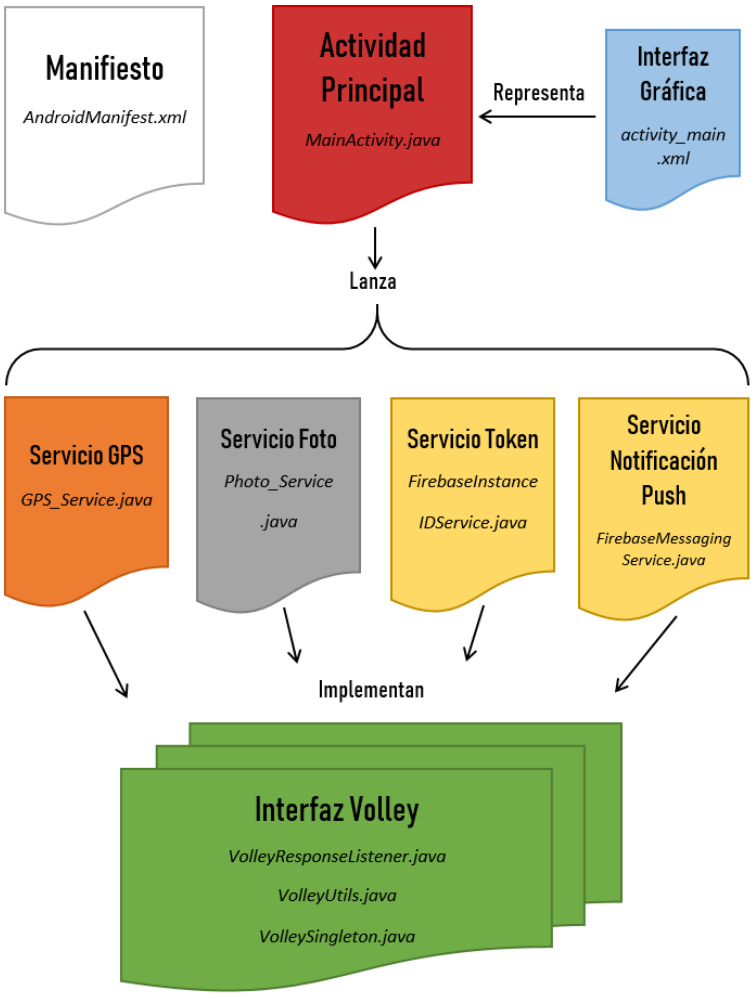

Figura 7: Visión global de la organización de la aplicación GPS Volley Location

En los siguientes apartados se explica en concepto las funciones que desempeña cada componente de la app. Para información más detallada, consultar el Anexo V.

5.2 Actividad principal

De cara al público, la app puede parecer sencilla en concepto y todavía más en lo que atañe el diseño interfaz gráfica.

Figura 8: Pantalla principal de la aplicación GPS Volley Location

Como se puede observar en la figura, la pantalla principal se compone de pocos elementos. Siendo exactos, de cinco: tres *TextViews* (contenedores de texto) y dos botones. Un *TextView* es estático, el de bienvenida, y otros dos que no aparecerán hasta que se pulse algún botón o se reciba una notificación push. Uno indica el estado de los servicios de la aplicación y el otro muestra el código de regata a la que está vinculada y si está en curso o no. Los dos botones habilitan y deshabilitan los servicios.:

Aparte de lanzar los servicios, la actividad principal también tiene otra función, relacionada con los permisos. Hasta la versión Android 5.1 (API 22), la forma que tenía el sistema de solicitar permisos al usuario era pidiéndolos una única vez justo antes de instalar la app, o en el caso de que se demandaran más permisos, se volverían a pedir que el usuario los aceptara antes de producirse una actualización. Por lo tanto, la única manera que tenían los usuarios de revocar permisos era desinstalando la app. Sin embargo, esto cambió a partir de la versión 6.0 de Android (API 23), dónde los permisos son controlados cada vez que la aplicación se ejecuta, permitiendo así que el usuario pueda revocar los permisos cuando quiera.

Como nuestra aplicación fue creada con la intención de poder funcionar en dispositivos con ambas versiones de software, tenemos que manejar las dos opciones, y el código que controla esta casuística también se encuentra en la actividad principal.

5.3 Servicios

Pese a la imagen simple de la aplicación, el código detrás de ella no resulta tan sencillo. Esto se debe esencialmente a que la aplicación trabaja con **servicios**. Son componentes de la aplicación que permiten realizar operaciones de larga ejecución en segundo plano, sin necesidad de una IU, por lo que "no se ven" dentro de la app. Son iniciados por un elemento de la aplicación y seguirán en funcionamiento aunque se cambie de actividad, aplicación o incluso se bloquee el dispositivo.

5.3.1 Tipos de servicios

En Android existen dos tipos de servicios:

- **Servicio iniciado:**

  Son iniciados por otra instancia de la aplicación mediante la llamada *startService()*. Permite que el servicio se ejecute en segundo plano de forma indefinida, incluso aunque el propio elemento que empezó el servicio sea destruido. Suele utilizarse para operaciones que se realicen una sola vez y no devuelvan el resultado al emisor, como por ejemplo descargar o cargar un archivo. Cuando terminan la tarea, deben detenerse por sí mismos.

- **Servicio de enlace:**

  Este servicio también se lanza mediante un componente de la aplicación con la llamada *bindService()* y se vincula él. A diferencia del anterior, este servicio ofrece una interfaz cliente-servidor que permite la interacción con el servicio, pudiendo enviar solicitudes y obtener resultados. Cuando se destruyen el objeto u objetos a los que se encontraba enlazado el servicio, éste también es destruido.

Dado que nuestra aplicación busca utilizar un servicio que se ejecute de forma ininterrumpida durante un periodo extenso de tiempo y no necesitamos más interfaz que la que se muestra en la figura anterior, utilizaremos un servicio iniciado, usando el método *startService()*. Aún así, aunque en la documentación de Android se abordan los servicios por separado, es posible crear un servicio que utilice funciones específicas de cada tipo.

5.3.2 Servicios implementados

Dentro de App Regatas tenemos cuatro servicios distintos que corresponden a cuatro funcionalidades distintas.

- **GPS_Service:**

  Se encarga de realizar la captura de datos de posición, en forma de latitud y longitud, la velocidad, el rumbo y el tiempo en el que son capturados. Contienen un *timer* o temporizador que asegura que los datos se mandan a una frecuencia de un envío por segundo, siempre y cuando se tenga la validación del comienzo de la regata.

- **Photo_Service:**

  Segundo servicio que contiene el código necesario para trabajar con la cámara del dispositivo, capturar la imagen, convertirla a Base64 y mandarla. Implementa la nueva API de Google (*camera2*) que gestiona las cámaras del dispositivo con un conjunto de métodos más completo respecto a la API anterior, ya obsoleta. En este caso el *timer* está configurado para mandar la imagen a razón de un envío cada 5 segundos, con idea de no saturar la base datos, ya que la imagen es el tipo de datos más pesado que se maneja en el proyecto.

- **FirebaseInstanceIDService:**

  Servicio que básicamente tiene la función de generar y mandar a la base de datos los *tokens*, identificación única del cliente de cara a la aplicación.

- **FirebaseMessagingService:**

  Por último, este servicio permite a la aplicación recibir las notificaciones push en sí. Recibe el mensaje mandado desde la nube del servicio FCM de Google, capta la información que puedan llevar asociada, y construyen la notificación que se muestra en el dispositivo. Es el responsable de transmitir a la actividad principal la señal de inicio/final de envío de datos.

5.4 Permisos implementados

Para el correcto funcionamiento de nuestra aplicación necesitamos el uso de varios permisos, tanto *normales* como *riesgosos*.

- **INTERNET:**

  Permiso normal. Permite a la aplicación acceder a sockets de red abiertos. Necesario para poder comunicarnos con la base de datos.

- **ACCESS_FINE_LOCATION:**

  Permiso riesgoso. Concede acceso a los datos de posición del dispositivo de forma precisa. La esencia del propósito del proyecto.

- **CAMERA:**

  Permiso riesgoso. Habilita el uso de las cámaras del dispositivo. Junto con los datos de posición nos permitirá dar al usuario final un reporte más completo de la situación actual de la embarcación.

5.5 Manifiesto implementado

Básicamente el manifiesto de GPS Volley Location contiene el nombre del paquete Java, permisos que necesita, información general de la aplicación como: icono, nombre, actividad que se lanza al iniciar y por último, los servicios que utiliza.

6 APLICACIÓN APP REGATAS

Aplicación enfocada principalmente a la presentación de los datos de posición e imagen al usuario. También incluye la función "Juez" que permite mandar las notificaciones push para habilitar o deshabilitar el envío de datos de su app hermana, GPS Volley Location.

6.1 Estructura

Debido a que App Regatas tiene cuatro funcionalidades distintas, se optó por crear una aplicación con un panel lateral de navegación. Se trata de un panel situado en el borde

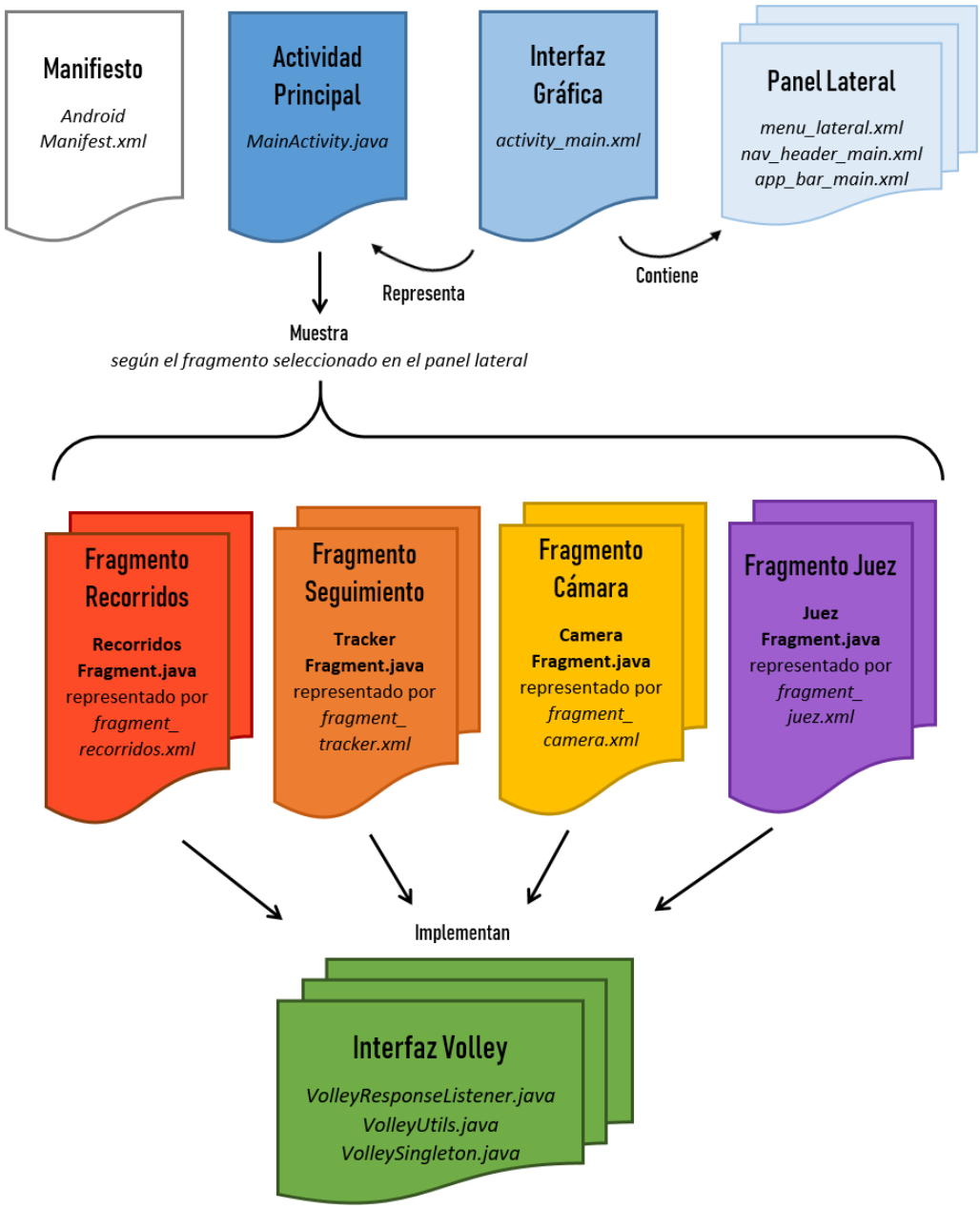

Figura 9: Esquema de funcionamiento de App Regatas

izquierdo de la pantalla que contiene las opciones de navegación de la app. La mayor parte del tiempo está oculto, pero aparece cuando se desliza un dedo desde el borde izquierdo hacia el centro de la pantalla o se toca el botón de menú en la barra de acciones.

Dependiendo de la opción seleccionada en el menú lateral se lanza una función distinta, las cuales se encuentran encapsuladas en instancias llamadas *fragments* o fragmentos, una especie de "subactividades". Como en el resto de la memoria, a continuación se

6.2 Actividad principal

Su función básica es controlar la navegación de la app de una funcionalidad a otra mediante el panel lateral y los *fragments*. Relaciona cada ítem del menú lateral con su correspondiente fragmento y lo vuelca en la pantalla. La interfaz de la actividad principal en sí no tiene mucho, sirve de contenedor de fragmentos. Se implementa en *activity_main.xml* mediante un *DrawerLayout*. Para implementar este tipo de diseño, *activity_main.xml* referencia otros tres archivos que ayudan a conformar el panel:

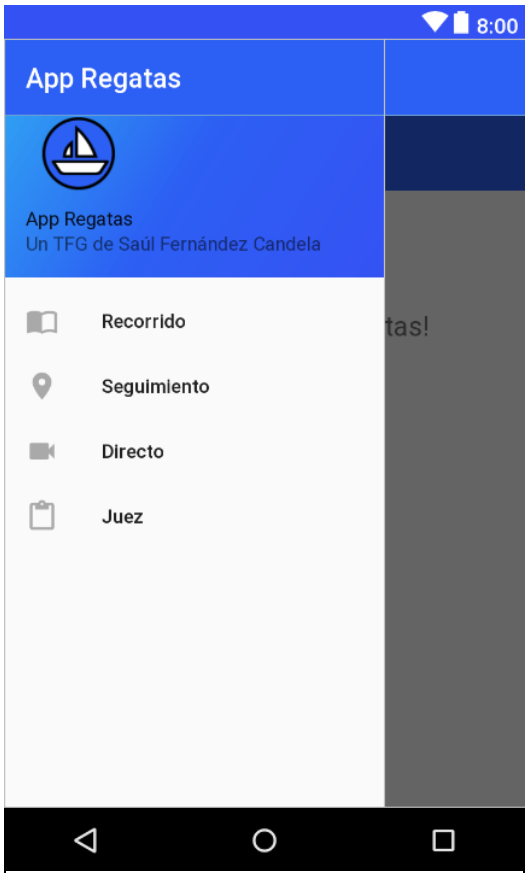

Figura 10: Aspecto final del panel lateral

- **app_bar_main.xml:**

  Fichero que sirve para instanciar la barra superior, en la que aparece el nombre de la app. Es necesaria para poder añadir en ella de forma programática el botón que abre el panel lateral.

- **menu_lateral.xml:**

  Contiene los títulos, iconos e identificación de las opciones de navegación que se muestran en el panel.

- **nav_header_main.xml:**

  Cabecera del panel lateral. Es opcional, su utilidad es puramente estética pero otorga a la aplicación mejor presencia.

6.3 Fragments

Los fragmentos fueron introducidos en Android 3.0 (API 11) principalmente para admitir diseño de IU más dinámicos y flexibles en pantallas grandes, como las de las tablets. Representan comportamientos o partes de la IU en una actividad, es decir, forman parte de ellas y por lo tanto, el ciclo de vida viene directamente dado por el de la actividad anfitriona. Sin embargo esto no quiere decir que el estado de un fragmento tenga que ser siempre el mismo que el de su actividad. Por ejemplo, cuando la actividad está pausada, también lo están todos sus fragmentos, y cuando la actividad se destruye, lo mismo ocurre con todos los fragmentos. No obstante, mientras una actividad se está ejecutando, estado *onResume()*, los fragmentos son manipulables de forma independiente. Pueden ser añadidos o eliminados, y además como cuentan con una pila de actividades, esta manipulación se puede deshacer presionando el botón *Atrás*.

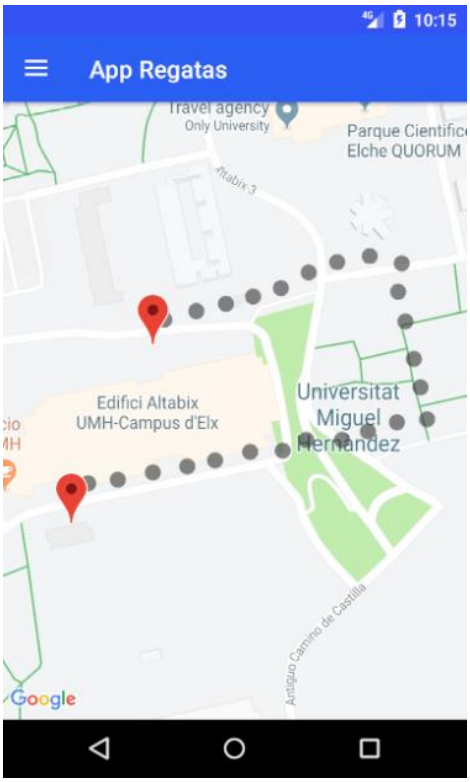

Figura 11: Aspecto final de fragment_recorridos.xml

Por otro lado, sí que tienen su propia IU, que no depende de la actividad y por lo tanto tienen sus *layouts* independientes para definirlas. Se pueden combinar múltiples fragmentos en una sola actividad para crear una IU multipanel y volver a usar un fragmento en múltiples actividades, dicho de otra forma, admite modularidad.

En nuestro caso, no hemos elegido trabajar con *fragments* para poder crear una IU específica para tablets, sino porque es la forma canónica de manejar un panel lateral, básicamente.

6.3.1 Fragments implementados

- **RecorridosFragment.java:**

En este fragment se recogen todas las posiciones recorridas por la embarcación durante el transcurso de la competición y se dibujan en el mapa, marcando el punto de inicio y final, como se observa en la figura 9. Para ello se recaban todos los datos de posición desde la API correspondiente en un array JSON que se procesa para crear un marcador por cada elemento. Como es evidente, para situar los marcadores es necesario un mapa, en Android, un *MapView*. Para poder utilizar estas vistas es necesario implementar la API de Google Maps, en el anexo correspondiente a esta app se encuentra el proceso que hay que seguir para ello.

- **TrackerFragment.java:**

Fragmento que se encarga de mostrar únicamente la última posición recibida. Al ser solo una posición la que se muestra en pantalla, se aprovecha para dar más datos sobre ella: se orienta el icono del barco hacia el rumbo captado y si se pulsa el marcador, aparecerá un pequeño bocadillo que nos indica la velocidad a la que navega en ese momento. Está pensada para ir refrescándose de forma automática cada segundo, por lo que contiene un temporizador del mismo tipo que los incorporados en los servicios de GPS Volley Location. También cuenta con un botón que sirve para refrescar la posición a placer, por si la espera de 1 segundo se nos hace larga o queremos comprobar si el servicio se ha quedado colgado o no.

Figura 12: Aspecto final de fragment_tracker.xml

- **CameraFragment.java:**

  Funcionalidad centrada en mostrar la última imagen captada desde la embarcación. Igual que el fragmento anterior, está pensado para ser consumido en directo, es decir, mientras la competición está en curso (hay envío de datos por parte de la app GPS Volley Location) . De hecho, si se consulta fuera de dicho tiempo, en vez de una imagen la aplicación mostrará un *TextView* diciendo "Servicio Offline". Se refresca a la misma razón que es mandada desde GPS Volley Location, una vez cada 5 segundos.

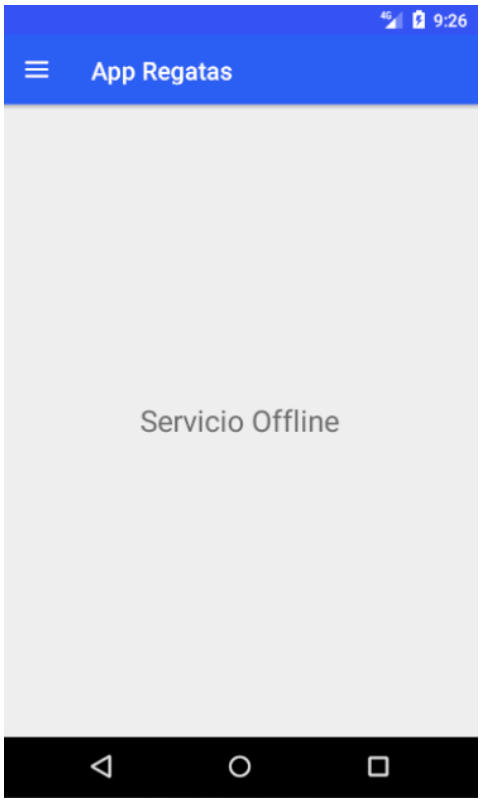

Figura 13: Aspecto de fragment_camera.xml con el servicio offline

Figura 14: Aspecto de fragment_camera.xml con el servicio online (imagen de ejemplo)

- **JuezFragment.java:**

  Encargado de dar inicio y final a las competiciones. Contiene una vista de tipo *EditText* que permite al usuario escribir información, en este caso el código identificativo de la regata, y dos botones: uno para mandar la notificación push como inicio y otro que manda la notificación que da el final.

Figura 15: Aspecto final de fragment_juez.xml

6.4 Permisos implementados

Pese a que la aplicación utiliza mapas, en nuestro caso no se necesita el permiso de acceso a la localización del usuario ya que realmente solo se utiliza la vista de un mapa para poner marcadores sobre él. Estos marcadores se obtienen mediante una petición a la API y no pertenecen al cliente de App Regatas, sino al de GPS Volley Location, por lo que pedirlos en la app es innecesario. El único permiso necesario para el funcionamiento es:

- **INTERNET:**

  Permite a la aplicación acceder a sockets de red abiertos. Se trata de un permiso normal, por lo que no hay que pedir permiso al usuario ni por lo tanto implementar el código que maneja la casuística de los permisos dependiendo de la versión de Android.

6.5 Manifiesto implementado

Pese a que conceptualmente, App Regatas es una aplicación más completa respecto a GPS Volley Location, su manifiesto es si cabe, más reducido.

El fichero incluye el nombre del paquete, el permiso, datos generales como el nombre de la app e icono que le da imagen, y la única actividad que aloja a los fragments. Estos últimos no necesitan ser declarados en el manifiesto. Al final del todo se declaran metadatos necesarios para el uso de la API de Google Maps. Sirven para identificar nuestra app de cara a Google.

7 PRESUPUESTO

A modo de estimación se expondrán los gastos que harían falta para llevarlo a cabo, pues las aplicaciones, el servidor, servicio web, base de datos y demás son un prototipo y no conforman un producto cerrado. Aún así, a continuación se exponen los gastos que hemos considerado oportunos:

- **Smartphone:**

  Dispositivo que cumpla con las especificaciones necesarias para poder correr las aplicaciones.

- **Tarifa operador de datos:**

  Necesaria para el envío de información desde los dispositivos móviles. Se considera que con 500 Mb por dispositivo es más que suficiente.

- **Tarifa servidor web:**

  Alojará nuestra API y base de datos.

- **Tarifas Google:**

  Solo será necesario pagar la cuota de publicación en Google Play, ya que el servicio de la plataforma de Google Maps que utilizamos (mapas estáticos) se puede usar sin ningún tipo de cobro. Por otro lado, Firebase ofrece el envío de 125.000 notificaciones push al mes en su plan gratuito, volumen suficiente para nuestro proyecto.

- **Equipo de desarrollo:**

  Equipo encargado del diseño y programación de las aplicaciones, API, y base de datos. Precio orientativo basado en consultas a empresas de este sector.

Concepto	**Precio (€)**
Smartphone Android Lollipop	35,06
Tarifa datos 500 Mb	2,50
Tarifa servidor web	130,00
Tarifas Google	21,92
Equipo desarrollo	20.000,00
Total presupuesto	20.189,48

8 CONCLUSIONES

Una vez desarrolladas las partes que conforman el proyecto, la valoración general es positiva. El sistema planteado es viable desde el punto de vista técnico, por lo que es apto para ser desarrollado a fondo.

Desde el punto de vista del trabajo ya realizado, podemos decir que todos los objetivos que se tomaron en un principio han podido ser alcanzados, ya sea con mayor o menor facilidad. Abarcando desde funciones simples como habilitar el uso de los sensores de posición del dispositivo para su captación, hasta otras más complejas como capturar y mandar imágenes hechas por la cámara mientras la pantalla se encuentra apagada, o enviar notificaciones de una aplicación a otra que contengan datos significantes para el código.

Por otro lado, si echamos la mirada hacia adelante, existe todavía más trabajo. Relacionado con el mantenimiento del servidor, un posible punto a desarrollar sería cómo manejar un volumen mucho mayor de peticiones, o cómo almacenar todos los recorridos de todas las embarcaciones de la forma más eficiente, así como también sería necesario ahondar en el aspecto de la seguridad del propio servidor. En cuanto a las aplicaciones Android, estaría más enfocado a implementar nuevas funcionalidades como pueden ser la identificación de usuarios mediante códigos QR, incorporar múltiples usuarios en la visualización en directo del seguimiento, o pensando en el ámbito de formación náutica, añadir sensores externos que proporcionen al entrenador información en el momento sobre la situación de la embarcación, para así ayudar a tecnificar la sesión de entrenamiento.

Si bien es cierto que el proyecto se centra en el desarrollo de código, podemos decir que se trata de un trabajo que engloba diversas tecnologías con distintos lenguajes de programación. Tenemos la plataforma Android, que trabaja con el lenguaje Java y XML y nos permite, con un dispositivo relativamente barato, acceder a un abanico enorme de información y funcionalidades. Tenemos el servidor Debian, que funciona con el lenguaje PHP y nos permite alojar, tratar y comunicar información de forma remota. Y tenemos Laravel, un *framework* también en PHP que junto a la base de datos, con su propio lenguaje, SQL, nos permite comunicarnos con las aplicaciones Android y almacenar sus datos. En conclusión, aunque formalmente el grueso del trabajo haya sido escribir código,

el proyecto resultante es la sinergia y potencia de la tecnología de la que disponemos, una imagen que describe acertadamente la labor de la ingeniería.

9 ANEXOS

Aquí encontraremos las especificaciones en detalle de los procesos de instalación e implementación de todos los componentes y herramientas utilizadas durante el desarrollo del trabajo. También tienen lugar las explicaciones del código detrás de las funcionalidades, servicios, APIs y aplicaciones que hacen posible el funcionamiento de cada una de las partes del proyecto. Nótese que para favorecer la lectura y las aclaraciones sobre los scripts, el código fuente de las aplicaciones y la API no se encuentran de forma completa e ininterrumpida en estos anexos. Sí se puede acceder a ellos en los archivos adjuntos en el CD, separados en las siguientes categorías: *Aplicación GPS Volley Location*, *Aplicación App Regatas*, y *Laravel*.

9.1 Anexo I: Instalación de Laravel

En esta instalación se da por hecho que ya hemos instalado tanto Debian como XAMPP. Ahora, para poder utilizar Laravel es necesaria la previa instalación del administrador de dependencias Composer en nuestro servidor. Para ello, debemos abrir una ventana de símbolo del sistema en el directorio donde queramos que Composer funcione y ejecutar las siguientes cuatro líneas de comando:

```
php -r "copy('https://getcomposer.org/installer', 'composer-
setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') ===
'93b54496392c062774670ac18b134c3b3a95e5a5e5c8f1a9f115f203b75bf9a129d5d
aa8ba6a13e2cc8a1da0806388a8') { echo 'Installer verified'; } else {
echo 'Installer corrupt'; unlink('composer-setup.php'); } echo
PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

El siguiente paso es instalar Laravel via Composer, utilizando de nuevo el símbolo del sistema. Los requisitos para nuestro servidor en la versión de Laravel que utilizamos son:

- Versión de PHP 7.0.0 o superior
- Tener habilitadas las siguientes extensiones:
    - OpenSSL PHP
    - PDO PHP
    - Mbstring PHP
    - Tokenizer PHP

- XML PHP

Podemos consultar el estado de las extensiones examinando el archivo *php.ini* ubicado en */opt/lampp/etc*. Si la extensión no está comentada (precedida por el símbolo de punto y coma), estará habilitada. Nótese que para hacer los cambios efectivos después de modificar el archivo *php.ini* debemos reiniciar el servidor web (XAMPP).

Una vez comprobado que el servidor cumple los requisitos de Laravel, nos situaremos en el directorio deseado para crear el proyecto de Laravel y ejecutaremos:

```
composer create-project --prefer-dist laravel/laravel miproyecto
```

Esto creará en la ubicación un directorio llamado "proyecto" contenedor de todas las dependencias de Laravel. Nótese que en la instrucción anterior, la última *miproyecto* palabra es la que sirve para dar nombre a nuestra aplicación.

En XAMPP, el proyecto se debe crear necesariamente en la dirección "/opt/lampp/htdocs/".

Finalmente, el último paso para llevar a cabo la instalación es configurar el archivo *".env"*. En algunos casos se puede encontrar como *".env.example"*. Si se da esta situación, debemos renombrarlo como *".env"*. Se trata de un script ubicado en la carpeta raíz del proyecto y que define el *entorno* en el que se encuentra (de ahí la extensión *".env"*, del inglés, *environment*). Para nuestro proyecto solo cambiaremos de momento la configuración relacionada con la conexión a la base de datos.

Los campos que lo controlan son:

```
DB_CONNECTION = mysql   // Para especificar el tipo de BD
DB_HOST = 127.0.0.1          // Dirección IP de la BD
DB_PORT = 3306          // Puerto IP de la BD
DB_DATABASE = apiregatas     // Nombre de la BD
DB_USERNAME =     root       // Nombre de usuario admin. de la BD
DB_PASSWORD =     <contraseña>      // Contraseña del admin. de la BD
```

Para comprobar que la instalación ha sido satisfactoria, podemos visitar la página de bienvenida que se encontraría añadiendo al final de la extensión de nuestro dominio "*/<nombre de mi proyecto>/public*".

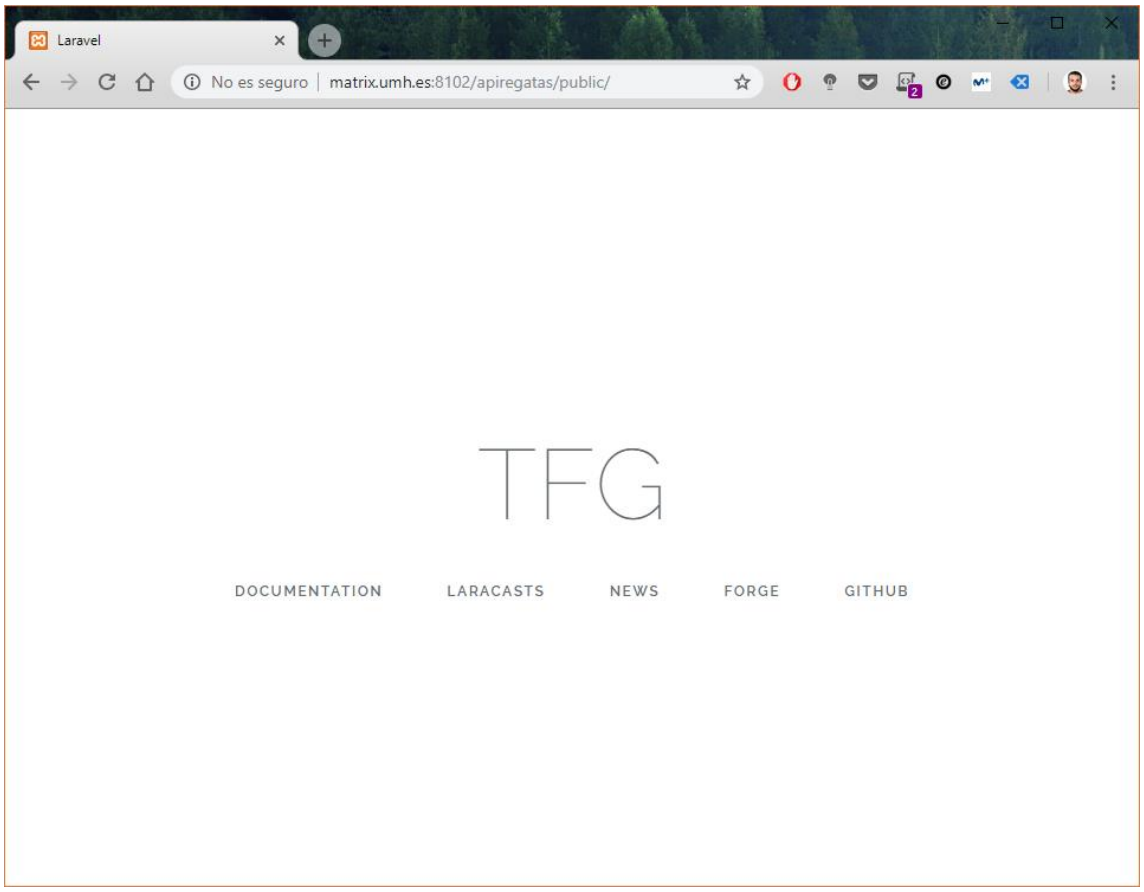

Figura 16: Página de bienvenida de Laravel

9.1.1 Errores

Es posible que durante la instalación y el resto del desarrollo de las APIs surjan problemas, a continuación se explicarán los que enfrentamos nosotros y cómo los resolvimos. En el caso de que se dé un error distinto, la mejor opción siempre es buscar el error que se ha reproducido en la web.

9.1.1.1 Acceso denegado

Para operar con Laravel, este necesita tener acceso de lectura y escritura en ciertos directorios. Concretamente, a los directorios *bootstrap* y *storage*. La solución aplicada fue ejecutar:

```
chmod -r 775 /storage
chmod -r 775 /bootstrap/cache
```

9.1.1.2 Clave de aplicación

Se trata de una cadena de caracteres generada de forma aleatoria que proporciona seguridad a nuestra aplicación. En principio su creación es automática y simultánea a la del proyecto, pero por distintos motivos puede quedarse vacía o defectuosa. La solución viene dada con la ejecución del comando:

```
php artisan key:generate
```

9.1.1.3 MariaDB

Como trabajamos con la distribución GNU GPL de MySQL, MariaDB, surgió un error relacionado con este sistema gestor de base de datos relacionado con la longitud por defecto de las cadenas de caracteres. Se enmendó añadiendo al script AppServiceProvider.php de Laravel, situado bajo *app/Providers* las siguientes líneas:

```
// En la cabecera:
use Illuminate\Support\Facades\Schema;

// Dentro de la función boot()
Schema::defaultStringLength(191);
```

9.1.1.4 Throttle

Laravel precisa de un limitador de número de peticiones HTTP que puede recibir por unidad de tiempo. Si este límite es sobrepasado hace devolver al servidor un error, haciendo infructuosa la petición. Puesto que nuestra aplicación recibe un gran número de peticiones, este error dificultaba el proceso de desarrollo. La solución adoptada fue comentar la siguiente línea del archivo *Kernel.php* ubicado en *app/Http*:

```
<?php

//...

protected $middlewareGroups = [
            'api' => [
            // 'throttle:60,1', <-- Línea comentada
            'bindings',
      ],
];
```

Analizando la línea que hemos deshabilitado, podemos ver que los parámetros del limitador corresponden a 60 peticiones en un periodo de 1 minuto. Es evidente que desactivar de forma total un limitador no es la opción más saludable para el servidor, pero

teniendo en cuenta que nos encontramos un entorno de depuración y prototipado a pequeña escala, podemos tomarnos esta licencia. En una instalación en producción habrá que analizar los valores apropiados de este parámetro para poder dar servicio al número de solicitudes estimado.

9.2 Anexo II: Plugins Laravel

Por si no estamos familiarizados con el concepto de "plugin" o complemento, se trata de un paquete adicional a un programa, normalmente desarrollado por terceros, que añade funcionalidades específicas a la aplicación principal, que en este caso es Laravel.

9.2.1 Reliese Laravel

Ciñéndonos a Reliese Laravel y traduciendo directamente de su repositorio en Github (https://github.com/reliese/laravel), Reliese Laravel es una colección de componentes de Laravel que buscan ayudar al proceso desarrollo de aplicaciones Laravel proporcionando capacidades generadas por código convenientes. Solo es compatible con MySQL y versiones de Laravel 5.1 o superiores. Al trabajar con este gestor de base de datos y Laravel 5.5, no tendremos ningún problema.

Para instalar este componente, lo haremos con la siguiente línea de comando:

```
composer require reliese/laravel
```

Una vez que se hayan descargado las dependencias del plugin a nuestro proyecto, debemos configurar varias cosas antes de poder importar los modelos de nuestra base de datos. La primera es asegurarnos de que la base de datos está correctamente configurada en los archivos *config/database.php* y *.env*.

El siguiente paso es añadir la siguiente línea en el archivo *config/app.php*:

```
// ...
// Dentro del array 'providers'

'providers' => [
    /*
     * Package Service Providers...
     */
    //Añadimos:
    Reliese\Coders\CodersServiceProvider::class,
],
// ...
```

Una vez hecho esto, guardamos los cambios y ejecutamos:

```
php artisan vendor:publish –tag=reliese-models
php artisan code:models
```

La primera línea sirve para que el Laravel "lea" los recursos instalados por Reliese, y la segunda importar el esquema de la base de datos a los modelos. Justo después de ejecutar

estas instrucciones deberán aparecer los archivos correspondientes a cada modelo bajo un nuevo directorio: *app/Models*.

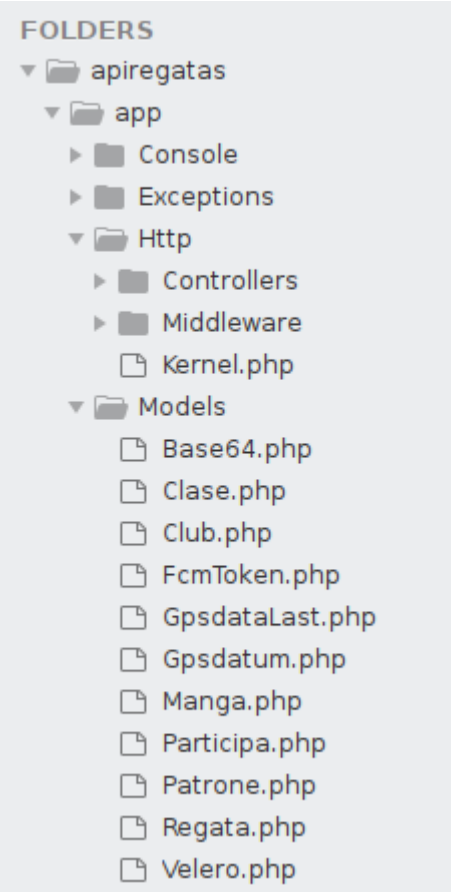

Figura 17: Modelos generados por el plugin Reliese

9.2.2 Laravel-FCM

Se trata de un paquete fácil de usar que hace posible enviar notificaciones push por Laravel a través de Firebase Cloud Messaging (FCM). Ahora bien, ¿qué es una notificación push? Y ¿qué es Firebase Cloud Messaging?

Las notificaciones push, como su propio nombre indica, son un tipo de notificaciones que se caracterizan por no tener su origen en el cliente en sí. Esto es, la notificación es enviada desde otra máquina, normalmente un servidor. De ahí el "push", la notificación es "empujada" desde otro dispositivo al que lo recibe.

Firebase Cloud Messaging, antes conocido como Google Cloud Messaging (GCM), es la plataforma que Google pone a disposición de desarrolladores su servicio de mensajes y notificaciones para Android, iOS y aplicaciones web. Cuenta con planes de suscripción en función del nivel de recursos que se necesiten, pero para el proyecto que nos ocupa, las herramientas que ofrece el plan gratuito son más que suficientes. El proceso de configuración de Firebase es bastante sencillo, tan solo tenemos que acceder con nuestra cuenta de Google a https://console.firebase.google.com y añadir el proyecto de Android en el que queremos implementar el servicio. En la propia página se explican los pasos de forma clara y dispone de cantidad de información para aclarar dudas, por lo que no se hará más hincapié.

Centrándonos de nuevo en el componente de Laravel, el primer paso para instalarlo es ejecutar la siguiente línea de comando en el directorio del proyecto:

```
composer require brozot/laravel-fcm
```

Acto seguido, debemos registrar las siguientes líneas dentro del fichero *config/app.php*:

```
// ...
// Dentro del array 'providers'

'providers' => [
    /*
     * Package Service Providers...
     */
    // Primera línea:
    LaravelFCM\FCMServiceProvider::class,
],

// Y dentro del array 'aliases'
// ...
'aliases' => [

//... Estas dos líneas:
       'FCM'     => LaravelFCM\Facades\FCM::class,
       'FCMGroup' => LaravelFCM\Facades\FCMGroup::class,
]
```

Después, para hacer efectiva esta última modificación ejecutamos:

```
php artisan vendor:publish
```

Por último para terminar la configuración, debemos introducir en el archivo *.env* la clave del servidor y la identificación del remitente que podemos encontrar en la configuración del proyecto en la página de Firebase, en la pestaña de Mensajería en la nube.

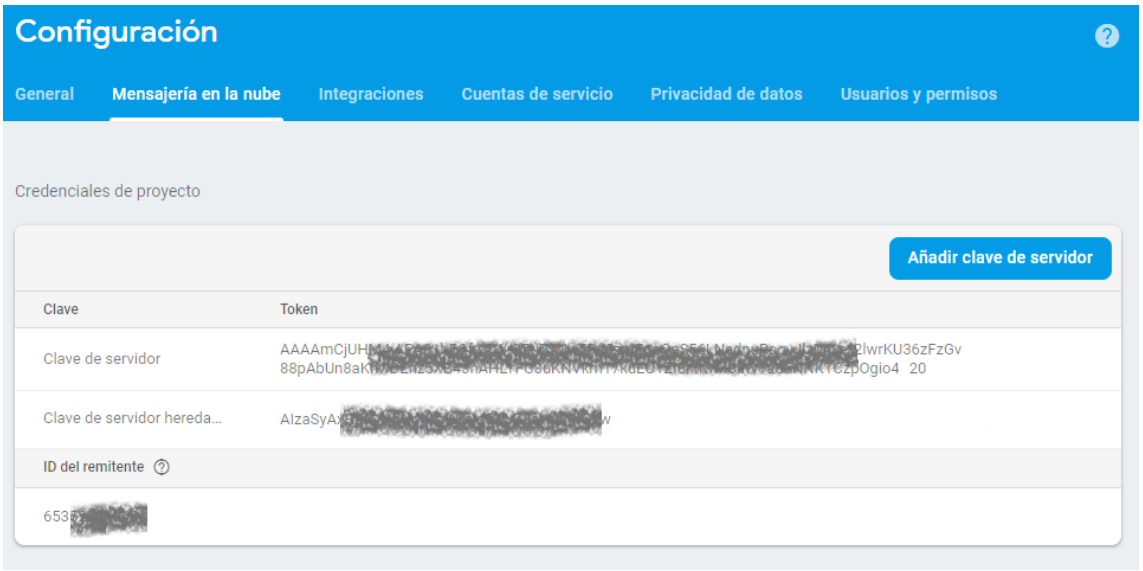

Figura 18: Panel de configuración donde se encuentran las claves necesarias para configurar Laravel-FCM

```
// ...
// Al final del todo añadimos:

FCM_SERVER_KEY = <clave del servidor>
FCM_SENDER_ID = <ID del remitente>
```

Ahora nuestro proyecto de Laravel ya estará preparado para mandar notificaciones push. Podemos ver las funciones utilizadas de este complemento explicadas en el Anexo III, pero si queremos ver todas las posibilidades que ofrece podemos consultar su página del repositorio online en https://github.com/brozot/Laravel-FCM.

9.3 Anexo III: API Laravel

En esta sección se mostrará y explicará el código utilizado en las funciones de la API que permite la comunicación entre las apps y la base de datos. También aprovecharemos para esclarecer la organización de los plugins del anexo anterior (en naranja) junto con las funciones de la API (las peticiones POST en azul, GET en verde) y su relación con las aplicaciones Android.

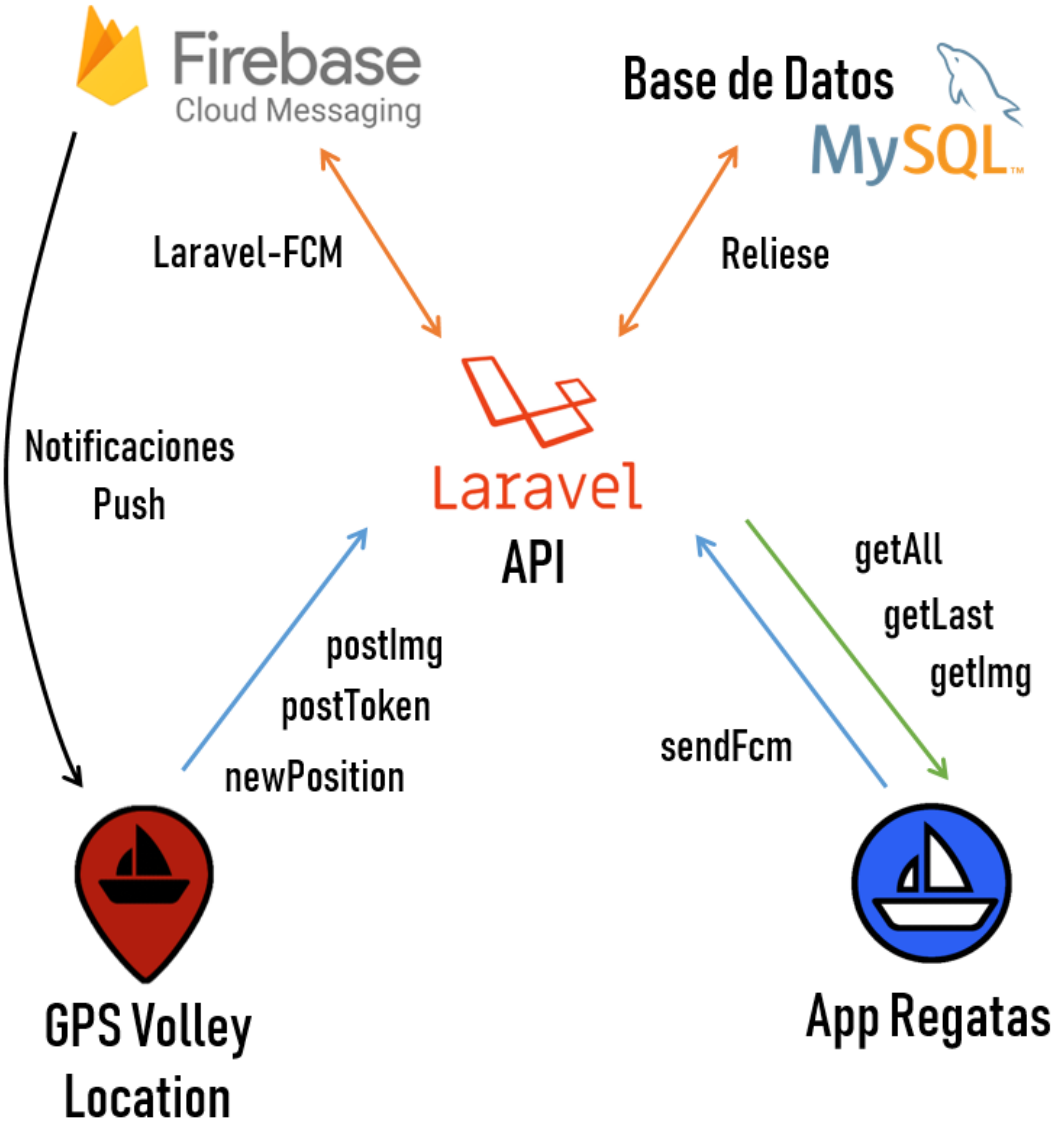

Figura 19: Diagrama de funcionamiento de la API con su entorno

Para facilitar la lectura, expondremos las funciones en el mismo orden que en la memoria.

9.3.1 Función newPosition

```
public function newPosition(Request $request)
{
      $position = Gpsdatum::create($request->all());

      $last_position = GpsdataLast::updateOrCreate(
            ['participa_id' => $request->participa_id],
            ['tiempo' => $request->tiempo,
             'latitud' => $request->latitud,
             'longitud' => $request->longitud,
             'rumbo' => $request->rumbo,
             'velocidad' => $request->velocidad]);
            return $request;
}
```

Antes de adentrarnos a explicar los métodos de Eloquent utilizados, debemos aclarar dos cosas. La primera es que esta función está diseñada para recibir un objeto JSON con la siguiente estructura:

```
{
    "participa_id": <int>,
    "tiempo": <bigint>,
    "latitud": <double>,
    "longitud": <double>,
    "rumbo": <float>
    "velocidad": <float>
}
```

La segunda es el método que se usa Laravel para acceder o modificar los registros de la base de datos. Utilizaremos la primera línea de código y unos colores a modo de ejemplo:

```
use App\Models\Gpsdatum

$position = Gpsdatum::create($request->all());
```

La primera parte marcada en verde, constituye la declaración de una variable PHP mediante el símbolo del dólar. Después del igual se produce la asignación a dicha variable, que se descompone en tres elementos. La palabra en azul corresponde al modelo. Debe estar incluido en la cabecera o de lo contrario el script nos dará un error al ejecutarlo. Seguido al modelo encontramos los dos puntos dobles, la forma de acceder a los métodos de Eloquent. En este caso el método utilizado, estilizado en naranja, es *create*. Nos permite en una sola línea de código insertar un nuevo registro en la tabla que está asociada al modelo. Como parámetros necesita los pares clave-valor que conforman la estructura de la tabla. Se trata de uno de los métodos de asignación masiva de Laravel y como tal, el modelo debe precisar en su definición de un atributo *fillable* o bien *guarded* para evitar vulnerabilidades. Ejemplo:

```
class Gpsdatum extends Eloquent
{
    protected $table = 'gpsdata';
    protected $primaryKey = 'tiempo';
    public $incrementing = false;
    public $timestamps = false;

    protected $ casts = [
        'participa_id' => 'int',
        'tiempo' => 'int',
        'latitud' => 'float',
        'longitud' => 'float',
        'rumbo' => 'float',
        'velocidad' => 'float'
    ];

    protected $fillable = [
        'participa_id',
        'tiempo',
        'latitud',
        'longitud',
        'rumbo',
        'velocidad'
    ]
}
```

Finalmente, el último término es el que se encuentra en negrita y representa los parámetros. En concreto, en esta línea de código que estamos comentando, hemos utilizado un método de las colecciones de Laravel (Clase propia de Laravel que implementa interfaces PHP y Laravel y son especialmente útiles para tratar arrays). El método *all()* de las colecciones nos devuelve todos los valores del array, por lo que eso es lo que pasamos como parámetros al método *create*.

En la segunda línea de código vemos como se repite el esquema de creación de un registro: variable PHP, modelo, dos puntos dobles, método y párametros. Si nos fijamos, ahora el modelo no es *Gpsdatum*, sino *GpsdataLast*. Utilizamos este modelo porque lo que buscamos en esta segunda línea es introducir un dato en una tabla distinta. El primer modelo iba dirigido a la tabla que almacena todas las posiciones del recorrido, mientras que el de ahora trabaja con la tabla que almacena la última posición captada. Por otro lado, pasando al método, podemos ver que es *updateOrCreate()*. Contiene dos parámetros. El primero es la condición que se evalúa para ver si se crea o se actualiza el registro, en nuestro caso, *participa_id*. El segundo son las claves que se actualizan con sus correspondientes valores: tiempo, longitud, latitud, rumbo y belocidad.

Finalmente, la última línea de código devuelve los datos que recibe la función de la API. Es el argumento de entrada de la función, contiene el cuerpo de la petición HTTP. En el

código se representa mediante la variable *request*. Su utilidad es básicamente de depuración, para confirmar el dato que se ha introducido sin error.

9.3.2 Función postImg

```
public function postImg(Request $request)
{
	$img = Base64::updateOrCreate(
		['participa_id' => $request->participa_id],
		['imagb64' => request->imagb64]);

	return $request;
}
```

Esta función recibe un JSON con dos claves, *participa_id* e *imagb64*. Un valor entero y otro una cadena de caracteres, respectivamente. Como el objetivo de esta función es solamente almacenar la última imagen, volvemos a utilizar el método que introdujimos en la función anterior: *updateOrCreate()*. No hay ninguna diferencia de su funcionamiento respecto al caso previo. Se evalúa si existe ya un registro con el valor de *participa_id* para actualizarlo con el segundo parámetro. Si no se encuentra ya en la tabla, se crea uno nuevo.

9.3.3 Función postToken

Función encargada de registrar el token generado por la aplicación en la base de datos.

```
public function postToken(Request $request)
{
	$token = new FcmToken;
	$token->participa_id = $request->participa_id;
	$token->token = $request->token;
	$token->save();
	return $request;
}
```

En esta función podemos ver un ejemplo distinto de cómo insertar un registro en una base de datos. Se reciben las claves *participa_id* y *token*, la primera ya recurrente en el anexo y la segunda una cadena de caracteres generada por el servicio FCM de Google. Este método de inserción resulta más convencional pero a su vez, supone más líneas de código. Se crea una nueva instancia del modelo *FcmToken*, se le asignan los campos correspondientes y por último se guarda la instancia, completándose así el registro en la tabla.

9.3.4 Función sendFcm

Función utilizada para generar y mandar la notificación push.

```
public function sendFcm(Request $request)
{
      $id = $request->regatas_id;
      $start = $request->start;

      $optionBuilder = new OptionsBuilder();
      $optionBuilder->setTimeToLive(60*5);

      if($start) {
                  $notificationBuilder = new
                  PayloadNotificationBuilder('Inicio de la regata');
            $notificationBuilder->setBody('¡La regata '.$id. ' ha
            comenzado!')->setSound('default');
      } else {
                  $notificationBuilder = new
                  PayloadNotificationBuilder('Fin de la regata');
            $notificationBuilder->setBody('¡La regata '.$id. ' ha
            termiando!')->setSound('default');
      }

      $dataBuilder = new PayloadDataBuilder();
      $dataBuilder->addData([ 'regatas_id' => $request->regatas_id,
                                    'start' => $start]);

      $option = $optionBuilder->build();
      $notification = $notificationBuilder->build();
      $data = dataBuilder->build();

      $tokens = DB::table('fcm')->pluck('token')->toArray();
      $downstreamResponse = FCM::sendTo($tokens, $option,
            $notification, $data);

      Return 'Notificación push enviada con éxito';
}
```

Sin duda esta es la función más extensa de todas, y en gran medida se debe a que trabaja con uno de los plugins que se presentan en el Anexo II: Laravel-FCM. Es decir, se trata de funciones que no han sido desarrolladas por el equipo detrás de Laravel, y como podemos observar no sigue tanto la filosofía Laravel de código fácil de leer y simple pero potente.

Pasando a desgranar el script, lo primero que hacemos es quedarnos con el código identificador de regatas *regatas_id* que nos llega en la petición junto a la variable booleana de comienzo/finalización *start*. Los siguientes pasos, si miramos un poco más hacia delante podemos darnos cuenta de que son constructores. Con ellos formaremos los parámetros que necesitamos para mandar la notificación. El primer constructor es *OptionsBuilder()* y mediante el método *setTimeToLive* nos permite especificar cuanto

queremos que la notificación se mantenga en el almacenamiento del servicio FCM si el dispositivo no está en línea. El segundo constructor, *PayloadNotificationBuilder*, determina el texto que aparecerá en la notificación: título y cuerpo. Dependiendo del valor de *start*, se indica si la regata va a comenzar ('1') o a terminar ('0'), y por lo tanto el texto deberá ser distinto en cada caso. Así, evaluamos *start* y construimos la opción correspondiente en cada caso con una sentencia *if-else*. El último constructor es *PayloadDataBuilder()* y sirve para introducir datos en forma clave-valor en la notificación. Lo utilizaremos para hacer que la notificación lleve los parámetros que recibimos en la petición a la API, *regatas_id* y *start*. Después se ejecuta la construcción llamando al método *build()* de cada constructor. Finalmente recogemos todos los *tokens* asociados a cada dispositivo en forma de array y con ellos junto al resto de parámetros que hemos construido (*$option*, *$notification* y *$data*) procederemos a mandar la notificación a los usuarios de la aplicación GPS Volley Location usando el método *sendTo*.

9.3.5 Función getAll

Función simple que proporciona todas las posiciones GPS almacenadas en la base de datos.

```
public function getAll()
{
      $positions = Gpsdatum::all();
      return $position;
}
```

En la primera línea recoge todos los registros de la tabla asociada al modelo *Gpsdatum* en la variable *$positions* y en la segunda se devuelve. Cogemos todos los registros sin ningún tipo de distinción porque en el proyecto trabajamos con un solo cliente.

9.3.6 Función getLast

Provee únicamente la última posición almacenada en la base datos.

```
public function getLast()
{
      $lastposition = GpsdataLast::first();
      return $lastposition;
}
```

Similar a la función *getAll()* en simpleza, pero con la diferencia de que en vez de recoger un array de valores solo recoge un registro, debido a que la tabla asociada al modelo solo guarda uno por cada *participa_id*. Como hemos mencionado antes también, al estar trabajando con un solo usuario, no hace falta que especifiquemos más en el código. De cualquier forma, hacerlo específico para cada participación en el código tendría fácil solución, añadiendo el código de identificación en la URL de la petición y añadiendo el método *where*(<clave>, <valor>) en la primera línea.

9.3.7 Función getImg

Provee la imagen guardada en la base de datos.

```
public function getImg()
{
	$img = Base64::first();
	return $img;
}
```

Función idéntica en concepto a *getLast()* y prácticamente también en código. Solo varía el modelo, que es el asociado a la tabla que almacena las imágenes codificadas en Base64 en una cadena de caracteres.

9.4 Anexo IV: Volley e interfaces

Si no pensamos utilizar Volley de forma puntual y queremos mandar peticiones a un servidor desde distintos scripts de nuestro código, la mejor práctica para evitar tener que instanciar una cola de peticiones para cada clase o servicio y los métodos para cada tipo de petición que tengamos que hacer es implementarlos de manera abstracta mediante una interfaz. En este anexo, veremos y comentaremos el código que hay detrás de dicha implementación.

9.4.1 Interfaz VolleyResponseListener

```
public interface VolleyResponseListener {
    void myOnResponse(String requestType, JSONArray response);
    void myOnResponse(String requestType, JSONObject response);
    void myOnError(String message);
}
```

Como se avanza en la memoria, el código propio de la interfaz no tiene mucho misterio. En el ejemplo vemos la interfaz implementada en la aplicación App Regatas, en la que se sobrecarga el método *myOnResponse* para poder trabajar tanto con respuestas JSON como con arrays de éstos. Para cerrar la interfaz tenemos el método que nos notifica las peticiones fallidas, *myOnError*. Todos los métodos de la interfaz llevan en el prefijo "*my*" para distinguirlos del método original de Volley.

9.4.2 Clase VolleySingleton

Clase *singleton*, esto es, clase que asegura que cuando el programa se ejecute, solo exista un único objeto de éste tipo. Nos resulta extramadamente útil porque sirve para unificar todas las peticiones que realiza Volley en una sola cola de peticiones.

```
public class VolleySingleton {
  // Clase Singleton:
  // Para obtener el contexto desde el que se llama a la
      interfaz y crear una cola de peticiones única

    private static VolleySingleton mInstance;
    private RequestQueue mRequestQueue;
    private static Context mCtx;

    private VolleySingleton(Context context)
    {
        mCtx = context;
        mRequestQueue = getRequestQueue();
    }
```

```
    public static synchronized VolleySingleton
      getInstance(Context context)
    {
        if (mInstance == null)
        {
            mInstance = new VolleySingleton(context);
        }
        return mInstance;
    }

    public RequestQueue getRequestQueue()
    {
        if (mRequestQueue == null)
        {
            mRequestQueue =
 Volley.newRequestQueue(mCtx.getApplicationContext());
        }
        return mRequestQueue;
    }

    public <T> void addToRequestQueue(Request<T> req)
    {
        getRequestQueue().add(req);
    }
}
```

Empecemos por las declaraciones y el constructor. La primera declaración es parte de la esencia del concepto *singleton*. El objeto *mInstance* declarado *static* que servirá para instanciar de forma única la cola. Seguido se declara la propia la cola de Volley que utilizaremos para realizar las peticiones y otro objeto estático de tipo *Context* que nos sirve para introducir el concepto de contexto en Android: Interfaz contenedora de información global sobre el entorno de la aplicación. Nos permite acceder a recursos y clases específicos, y como vemos en el código, es necesario para crear la cola de peticiones.

Pasando al constructor, podemos ver que es privado, para no permitir que se genere un constructor por defecto. En él se inicializan las variables de contexto, con el parámetro que recibe, y la cola mediante la llamada al método *getRequestQueue()*, que se encarga de generar y devolver el objeto de cola de peticiones mediante el contexto que acaba de ser instanciado, siempre que no haya sido creada antes (sentencia *if*).

El método getInstance conforma la otra parte de la esencia *singleton*. Es el creador sincronizado que evita la instanciación múltiples así como posibles problemas multihilo. Se evalúa si ya ha sido creado con anterioridad el objeto *mInstance* y si no es así, se crea el objeto con el contexto.

Por último, el método *addToRequestQueue()* recibe una petición como parámetro y se encarga de añadirla a la cola accediendo a ella mediante *getRequestQueue()*, y con el operador punto, a su método *add()*.

9.4.3 Clase VolleyUtils

Clase que contiene las utilidades de Volley. Todo el código que se concentra en esta clase, es el que nos ahorramos repetir en cada una de las actividades, fragmentos o servicios que queramos utilizar Volley. Es la clase más extensa de las tres ya que contiene las creaciones de todos los tipos de peticiones, no en cuanto a los métodos GET, PUT, DELETE y demás, pues este es un parámetro de la clase, sino en cuanto al tipo de objeto que devuelven. El código que vamos a desgranar pertenece también a App Regatas ya que es más rico en contenido. Lo comentaremos a partes para facilitar la lectura.

```
public class VolleyUtils {
   private WeakReference<VolleyResponseListener> listener;
   private Context theContext;
   private String requestDataType;
   private JSONObject jsonObj;
   private int rqMethod;
   private String url;
   private JSONArray jsonArray;

   public VolleyUtils(Context context, VolleyResponseListener
     listener, String requestDataType, JSONObject jsonObj,
     String url, int rqMethod) {
       this.listener = new WeakReference<>(listener);
       this.theContext = context;
       this.url = url;
       this.requestDataType = requestDataType;
       this.jsonObj = jsonObj;
       this.jsonArray = null;
       this.rqMethod = rqMethod;
   }

   public VolleyUtils(Context context, VolleyResponseListener
     listener, String requestDataType, JSONArray jsonArray,
     String url, int rqMethod){
       this.listener = new WeakReference<>(listener);
       this.theContext = context;
       this.url = url;
       this.requestDataType = requestDataType;
       this.jsonObj = null;
       this.jsonArray = jsonArray;
       this.rqMethod = rqMethod;
   }

//...
```

En este fragmento ocurren las declaraciones de las variables globales y la sobrecarga del constructor, en función de si la petición que se quiere hacer es de un objeto JSON o un array de éstos. En el primero el array JSON se declara nulo, y en el segundo es el objeto el que se inhabilita. En cuanto a las variables, aclararemos un par de casos ya que pueden resultar no tan obvios como el resto. La variable *listener* hace referencia a la interfaz *VolleyResponseListener* que tiene que recibir el constructor, pero tiene un detalle, que se trata de un tipo *WeakReference*. Esto hace básicamente que el objeto sea eliminado por el Garbage Collector para prevenir ocupar la memoria de forma inútil. La cadena de caracteres *requestDataType* es un identificador que nos servirá para distinguir las peticiones. Recordemos que al tratarse de la implementación de una interfaz, los métodos son abstractos, y si no les otorgamos ningún distintivo, los métodos pueden ser confundidos. Es decir, si yo no especifico en un principio de qué tipo de petición se trata, cuando reciba la respuesta tampoco voy a saberlo, por lo que no podré distinguir la forma de actuar para cada una.

```
//...
public void runJsonArrayRequest() {
JsonArrayRequest jsonArrayRequest = new
JsonArrayRequest(this.rqMethod, this.url, this.jsonArray,
new Response.Listener<JSONArray>(){
@Override
public void onResponse(JSONArray response) {
if (listener.get() != null) {
         listener.get().myOnResponse(requestDataType, response);
}
          }
          }, new Response.ErrorListener() {
                @Override
                public void onErrorResponse(VolleyError error) {
                    if (listener.get() != null){
                 listener.get().myOnError(error.toString());
      }
                }
            });

//Accedemos a la cola de petición a través de la clase Singleton
VolleySingleton.getInstance(this.theContext).addToRequestQueue(
jsonArrayRequest);
    }
//...
```

En el código anterior encontramos el método que hace posible las peticiones con arrays JSON de Volley. No se necesitan todos los parámetros del constructor, con la URL de la API, el método de petición HTTP, el array JSON y un *listener* para la respuesta (el cual se instancia en el momento), se monta la petición. Por último se añade a la cola a través

de la clase *singleton* de Volley y sus métodos *getInstance* y *addToRequestQueue* explicados en anterioridad.

Para terminar, tenemos la creación de una petición con un objeto JSON. Es prácticamente idéntico al método anterior, el único parámetro que cambia sería el array JSON por el tipo objeto.

```
//...
public void runJsonObjectRequest() {
JsonObjectRequest jsonObjectRequest = new
JsonObjectRequest(this.rqMethod, this.url, this.jsonObj,
    new Response.Listener<JSONObject>() {
        @Override
        public void onResponse(JSONObject response) {
            if (listener.get() != null) {
                listener.get().myOnResponse(requestDataType,
                  response);
            }
        }
    }, new Response.ErrorListener() {
        @Override
            public void onErrorResponse(VolleyError error) {
                if (listener.get() != null ) {
                    listener.get().myOnError(error.toString());
                }
            }
        });
        VolleySingleton.getInstance(theContext).addToRequestQueue(
          jsonObjectRequest);
}
```

9.4.4 Implementación

Tras la creación de la interfaz y las clases, debemos implementarlas en la actividad, servicio o *fragment* que queramos que haga las peticiones. Para ello, lo primero que debemos hacer es implementar la interfaz *VolleyResponseListener* en la clase y crear un objeto de tipo *VolleyUtils* de la siguiente manera:

```
Public class MiServicio implements VolleyResponse Listener {
      VolleyResponseListener volleyListener;
      VolleyUtils volleyUtils;

      public void onCreate() {
          volleyListener = this;
      }
}
```

Y más tarde, para hacer la petición y mandarla debemos instanciar el objeto *volleyUtils* con sus respectivos parámetros y acceder al método *runJSONObjRequest()* o bien *runJSONArrayRequest()*, dependiendo del tipo de objeto de la petición:

```
//Para una petición de objeto JSON
volleyUtils = new VolleyUtils(getApplicationContext(),
volleyListener,
requestType,  jsonObject,
"http://matrix.umh.es:8102/apiregatas/public/api/pos",
Request.Method.POST);
volleyUtils.runJSONObjRequest();
```

```
//Para una petición de array JSON:
volleyUtils = new VolleyUtils (getApplicationContext(),
volleyListener,
requestType, jArray,
"http://matrix.umh.es:8102/apiregatas/public/api/pos",
Request.Method.GET);
volleyUtils.runJSONArrayRequest();
```

9.5 Anexo V: Aplicación GPS Volley Location

En este anexo se mostrará de forma detallada el código fuente detrás de las funciones de la aplicación GPS Volley Location que no se ha enseñado en la memoria principal.

9.5.1 Actividad Principal

Se trata de un script que consta con 224 líneas, por lo que para poder abordarlo de una manera más cómoda, explicaremos el código de forma modular, método a método, obviando las cabeceras.

9.5.1.1 Layout

Fichero que describe mediante XML la apariencia que tendrá un elemento en la aplicación. No se trata de una interfaz demasiado compleja, de hecho es bastante simple, pero aún así puede tener más líneas de lo que parece:

```
<?xml version="1.0" encoding="utf-8"?>
<android.widget.RelativeLayout
    xmlns:Android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context =
      "sfc.tfg.com.gpsvolleylocationfinal.MainActivity">

    <TextView
        android:id="@+id/tvBienvenida"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp"
        android:text="¡Bienvenido a GPS Volley Location!"
        android:textAlignment="center"
        android:textColor="@android:color/black"
        android:textSize="20sp" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="125dp"
        android:text=""
        android:textAlignment="center"
        android:textColor="@android:color/darker_gray"
        android:textSize="25sp" />

<TextView
```

```
        android:id="@+id/tvData"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="175dp"
        android:text=""
        android:textAlignment="center"
        android:textColor="@android:color/darker_gray"
        android:textSize="25sp" />


 <Button
        android:id="@+id/btnStart"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="Start" />

    <Button
        android:id="@+id/btnStop"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignStart="@+id/btnStart"
        android:layout_below="@+id/btnStart"
        android:layout_marginTop="20dp"
        android:text="Stop" />

</android.widget.RelativeLayout>
```

9.5.1.2 onCreate

```
@Override
protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);
   setRequestedOrientation(ActivityInfo.
      SCREEN_ORIENTATION_PORTRAIT);
   setContentView(R.layout.activity_main);
   btn_start = findViewById(R.id.btnStart);
   btn_stop =  findViewById(R.id.btnStop);
   textView = findViewById(R.id.textView);
   tvData = findViewById(R.id.tvData);if(broadcastReceiver ==
      null){
    broadcastReceiver = new BroadcastReceiver() {
              @Override
              public void onReceive(Context context, Intent
             intent) {
                  Bundle extras = intent.getExtras();
                  HashMap hashMap = (HashMap) extras.get("data");
                  tvData.setText("Regatas_id: " +
                     hashMap.get("regatas_id")+",
                     Start: "+hashMap.get("start"));
                  if(hashMap.get("start").equals("1")){
                      start = true;
                      if (btnstart) {btn_start.performClick();}
                  }
                     else {
                         start = false;
```

```
                        btn_stop.performClick();
                    }
                }
            };
        }

    LocalBroadcastManager.getInstance(MainActivity.this).
     registerReceiver(broadcastReceiver,
     new IntentFilter("START_SIGNAL"));
}
```

Método en el que se configura el estado global del diseño de la aplicación. Lo utilizamos para fijar la interfaz (*activity_main.xml*) y bloquearla en modo retrato, ya que la app está pensada para que el móvil se encuentre orientado apuntando con la cámara al tripulante de la embarcación, de forma que pueda verse el estado del mismo remotamente. Después inicializamos las vistas que interactúan con el usuario e instanciamos el *BroadcastReceiver* que nos permitirá operar con las notificaciones push recibidas por el servicio *FirebaseMessagingService*. Dentro del *BroadcastReceiver* tenemos el código que mostrará la regata que va a iniciarse o terminar y en función de la variable que determina esto último, *start*. Si su valor es igual a 1 se activará la bandera local con el mismo nombre y se pulsará de forma programática el botón de inicio *btn_start*. En caso de que el valor no sea 1 la bandera *start* tomará el valor *false* y se pulsará el botón de finalización *btn_stop*. Por último, es necesario registrar el *BroadcastReceiver* que hemos declarado antes.

Se decide declarar aquí el *BroadcastReceiver* en vez de en el método *onStart()* porque así se abarca todo el ciclo de vida de la actividad y de esta manera la aplicación es capaz de comenzar y terminar el envío de datos cuando la aplicación se encuentre en segundo plano o con la pantalla apagada.

9.5.1.3 onDestroy

```
@Override
protected void onDestroy() {
    super.onDestroy();
    try {
         unregisterReceiver(broadcastReceiver);
    } catch (Exception e) {
         e.printStackTrace();
    }
}
```

Como hemos registrado el *BroadcastReceiver* en el método *onCreate()*, es en el método *onDestroy()* donde se debe borrar.

9.5.1.4 onStart

En este método se encuentra la comprobación de los permisos y las acciones que desencadenan los botones. Contiene la gran parte de código del script así como llamadas a otras funciones, por lo que descompondremos el método en las siguientes secciones:

- **Permisos:**

  Lo primero es comprobar en que versión de Android nos encontramos haciendo uso de la función *IsOlderPermissionVersion()*:

```
private boolean IsOlderPermissionVersion(){
    boolean older=false;
    int CurrentSDKVersion= Build.VERSION.SDK_INT;
    int MarshmallowSDKVersion=Build.VERSION_CODES.M;
    if (CurrentSDKVersion<MarshmallowSDKVersion) older=true;
    return older;
}
```

Como vemos, recogemos por un lado la versión en la que nos encontramos, la versión correspondente a Marshmallow y la comparamos. Si la versión del sistema es inferior, se devuelve la variable *older* como *true*, y en el caso contrario, si es superior, se devolverá como *false*.

Una vez conocida la versión en la que nos encontramos podemos plantear las acciones necesarias para pedir los permisos de la manera correcta.

```
if (IsOlderPermissionVersion()) {
    if (CheckPermission(Manifest.permission.INTERNET)&
      CheckPermission(Manifest.permission.
      ACCESS_FINE_LOCATION)&
      CheckPermission(Manifest.permission.CAMERA))
        flagPerms = true;
    else
        Toast.makeText(this, "You don't have permissions
          to run this action", Toast.LENGTH_LONG).show();
} else {
    if (CheckPermission(Manifest.permission.INTERNET)&
      CheckPermission(Manifest.permission.
      ACCESS_FINE_LOCATION)&
      CheckPermission(Manifest.permission.CAMERA)) {
        flagPerms = true;
    } else {
        if (!flagPerms){
            requestPermissions(new String[]{
              Manifest.permission.ACCESS_FINE_LOCATION,
              Manifest.permission.INTERNET,
              Manifest.permission.CAMERA}, REGATAS_CODE);
        } else {
            Toast.makeText(this,"Por favor, activa el permiso
              para poder realizar la llamada",
```

```
                Toast.LENGTH_LONG).show();
              Intent intentSettings = new Intent(
                Settings.ACTION_APPLICATION_DETAILS_SETTINGS);
              intentSettings.addCategory(Intent.CATEGORY_DEFAULT);
              intentSettings.setData(Uri.parse(
                "package:"+getPackageName()));
              intentSettings.addFlags(
                Intent.FLAG_ACTIVITY_NEW_TASK);
              intentSettings.addFlags(
                Intent.FLAG_ACTIVITY_NO_HISTORY);
              intentSettings.addFlags(
                Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
              startActivity(intentSettings);
          }
}
```

En esta parte del código se introduce la función *CheckPermission*, creada por nosotros mismos. Esta pequeña función recibe un permiso, comprueba si está declarado o disponemos ya de él y en caso positivo devuelve el valor *true*; *false* en caso contrario. También vemos por primera vez la variable binaria *flagPerms*, que se utilizará como bandera para marcar si los permisos han sido concedidos o no, y posteriormente, habilitar los botones que lanzan los servicios.

El primer paso del algoritmo contempla el caso de que nos encontremos en una versión antigua. Se llama a la función *CheckPermission* para comprobar si se tienen cada uno de los permisos, y en el caso de que todos hayan sido concedidos, la variable *flagPerms* se le da el valor *true*. Si falta algún permiso, *flagPerms* permanece en su estado por defecto de *false*.

Si nos encontramos en una versión más moderna, el código es un poco más largo ya que tenemos que comprobarlo cada vez que se ejecuta la app y no solo en la instalación/actualizaciones como en las versiones antiguas. Lo primero es comprobar los permisos de la misma manera que antes, si los tenemos ya concedidos se le asigna el valor *true* a *flagPerms* y aquí terminaría, pero todavía queda una casuística que contemplar. Si no tenemos los permisos puede ser porque han sido denegados, o porque se trata de la primera vez que se comprueban. Para evaluar esto se consulta el valor de *flagPerms*, si es *false* no han sido pedidos antes y se llama a la función de Android *requestPermissions*. Esta función recibe un array con los permisos que se piden, y un código que identifica la petición. El flujo de este caso continúa en el siguiente *callback*:

```
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull
String[] permissions, @NonNull int[] grantResults) {
    switch (requestCode) {
```

```
        case REGATAS_CODE:
            if (grantResults[0] ==
              PackageManager.PERMISSION_GRANTED &&
              grantResults[1] ==
              PackageManager.PERMISSION_GRANTED &&
              grantResults[2] ==
              PackageManager.PERMISSION_GRANTED) {
                  flagPerms = true;

            } else {
                Toast.makeText(this, "No se concedió el
                  permiso", Toast.LENGTH_LONG).show();
            }
            break;
        default:
            super.onRequestPermissionsResult(requestCode,
              permissions, grantResults);
            break;
    }
}
```

En este método que es llamado de forma automática por el sistema Android, se identifica el código de la petición mediante una estructura *switch*/*case* y dentro de este caso se obtienen directamente del sistema el estado de los permisos. Si todos han sido concedidos se da el valor *true* a la bandera *flagPerms* y en caso contrario se avisa al usuario mediante un *Toast* que los permisos no fueron concedidos.

Finalmente, volviendo al último caso pendiente, si los permisos en la versión moderna ya habían sido denegados, se insta al usuario a conceder los permisos de nuevo con un *Toast* y llevándolo a los ajustes mediante un *intent*.

- **Botones:**

  Controlan el inicio y final de los servicios. El uso de los botones se habilita mediante la variable *flagPerms*, de manera que si el usuario no ha otorgado los permisos necesarios para funcionar, la aplicación inhabilitará los botones y por lo tanto sus funciones.

```
if(flagPerms) {
    btn_start.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            btnstart = true;
            try {
                Intent iGps = new Intent(
                  getApplicationContext(), GPS_Service.class);
                Intent iPhoto = new Intent(
                  getApplicationContext(), Photo_Service.class);
                iGps.putExtra("start", start);
                iPhoto.putExtra("start", start);
                startService(iGps);
```

```
                startService(iPhoto);
                if (start && btnstart) {
                    textView.setText("Servicio Activado y
                      Enviando");
                } else{
                    textView.setText("Servicio Activado");
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });

    btn_stop.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Intent iGps = new Intent(getApplicationContext(),
              GPS_Service.class);
            Intent iPhoto = new Intent(getApplicationContext(),
              Photo_Service.class);
            btnstart = false;
            iGps.putExtra("start", start);
            iPhoto.putExtra("start", start);
            try {
                jsonObject = new JSONObject();
                jsonObject.put("participa_id", 1);
                jsonObject.put("imagb64", "stop");
                requestType = "imag";
                volleyUtils = new   VolleyUtils(
                  getApplicationContext(), volleyListener,
                  requestType, jsonObject,
                  getResources().getString(R.string.img),
                  Request.Method.POST);
                volleyUtils.runJSONObjRequest();
            } catch (JSONException e) {
                e.printStackTrace();
            }
            stopService(iGps);
            stopService(iPhoto);
            textView.setText("Servicio Desactivado");
        }
    });
}
```

En el botón *btn_start*, lo primero que se hace es activar la bandera de nombre análogo *btnstart*, que nos servirá para identificar el servicio como activado. Acto seguido lanzamos los servicios *GPS_Service* y *Photo_Service* pasándoles como parámetro la variable *start* recibida de la notificación. Finalmente, lo último que se hace es informar de en cuál de las dos posibles situaciones en las que se puede encontrar la aplicación nos encontramos cuando el botón de inicio es pulsado. Si tanto la variable *start* proveniente de la notificación push, como la de *btnstart* son *true*, significa que los servicios están activados y mandando datos por lo que se mostrará mediante un *TextView* "Servicio Activado y Enviando". Si no se da este caso, significa que la regata no ha empezado o ya

ha terminado, por lo tanto los servicios están activados pero no mandando datos, situación que se indica mostrando en pantalla “Servicio Activado”.

El siguiente botón, *btn_stop*, se encarga de parar los servicios, desactivar la bandera *btnstart*, y mostrar en pantalla “Servicio Desactivado” utilizando el mismo TextView del botón de inicio. Además, manda una señal a la función de captura de imágenes que permitirá a la aplicación App Regatas determinar que el servicio de imagen está parado y transmitirlo al usuario.

9.5.2 Servicios

En esta parte del anexo se mostrará el código fuente de los servicios, componentes capaces de realizar funciones en segundo plano, cruciales para esta aplicación.

9.5.2.1 Servicio GPS_Service

Servicio encargado de recoger la posición del dispositivo y mandarla a la base de datos. Para hacer esto posible implementa un *LocationListener* y un *LocationManager*, clases que permiten acceder a las funciones de localización del dispositivo y notificar los cambios que ocurren en los sensores que la recogen. También se implementa, para poder comunicar los datos a la base de datos, la librería Volley mediante la interfaz explicada en el anexo anterior.

9.5.2.1.1 onStartCommand

```
Método llamado por el sistema cada vez que se inicia el servicio.
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    start = intent.getExtras().getBoolean("start", false);
    return super.onStartCommand(intent, flags, startId);
}
```

Dentro de él es donde podemos recoger la información asociada al *intent* que inicia el servicio, por eso aquí guardamos en la variable global *start* el valor de la notificación push que actúa como interruptor para mandar o no la información a la base de datos.

9.5.2.1.2 onCreate

Método más extenso del servicio. Explicaremos su funcionamiento por partes.

```
public void onCreate() {
    volleyListener = this;
    listener = new LocationListener() {
```

```
        @Override
        public void onLocationChanged(Location location) {
            Double lat = location.getLatitude();
            Double lng = location.getLongitude();
            tiempo = location.getTime();
            Float rumbo = location.getBearing();
            Float vel = location.getSpeed();
            try {
                jsonObject = new JSONObject();
                jsonObject.accumulate("participa_id", 1);
                jsonObject.accumulate("tiempo", tiempo);
                jsonObject.accumulate("latitud", lat);
                jsonObject.accumulate("longitud", lng);
                jsonObject.accumulate("rumbo", rumbo);
                jsonObject.accumulate("velocidad", vel);
            }
            catch (Exception e) {
                Log.d("Error Creating JSON",
                  e.getLocalizedMessage());
            }
        }

        @Override
        public void onStatusChanged(String s, int i, Bundle
          bundle) {

        }

        @Override
        public void onProviderEnabled(String s) {

        }

        @Override
        public void onProviderDisabled(String s) {

            Intent i = new Intent(
              Settings.ACTION_LOCATION_SOURCE_SETTINGS);
            i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            startActivity(i);
        }
    };

    locationManager = (LocationManager) getApplicationContext().
      getSystemService(Context.LOCATION_SERVICE);

    //noinspection MissingPermission
    locationManager.requestLocationUpdates(
      LocationManager.GPS_PROVIDER,1000, (float) 0.5, listener);
    setGPSTimer();
    }
}
```

Lo primero que hacemos es diferenciar los *listeners*, el primero pertenece a Volley, y se encuentra implementado en el servicio, por eso lo instanciamos mediante el puntero *this*. El segundo nos notificará la localización del sistema. Para implementarlo debemos sobrescribir todos sus métodos, pero utilizaremos dos básicamente:

- **onLocationChanged :**

  Es llamado cuando se registra un cambio en la ubicación GPS. Dentro de él recogemos la latitud, longitud, tiempo, rumbo y velocidad registrada por el dispositivo y creamos un objeto JSON con ella.

- **onProviderDisabled:**

  Método pensado para controlar la inhabilitación de la ubicación por parte del usuario. Enviamos un *intent* dirigido a los ajustes de ubicación para que se habilite la ubicación cuanto antes posible, ya que es la piedra angular de la aplicación.

Justo después de declarar el *LocationListener*, se instancia el *LocationManager* y lo configuramos para que se actualice cada segundo (1000 ms en el código) o cada 0,5 metros, lo que ocurra antes.

Y para terminar el código en el método *onCreate()* hacemos una llamada a la función *setGPSTimer()*, escrita por nosotros:

```
public void setGPSTimer() {
    final Handler handler = new Handler();
    timer = new Timer();
    doVolleyRequest = new TimerTask() {
        @Override
        public void run() {
            handler.post(new Runnable() {
                public void run() {
                    try {
                        if (start) {
                            requestType = "gpsdata";
                            volleyUtils = new VolleyUtils(
                              getApplicationContext(),
                              volleyListener, requestType,
                              jsonObject, getResources().
                              getString(R.string.pos),
                              Request.Method.POST);
                            volleyUtils.runJSONObjRequest();
                        }
                    } catch (Exception e) {
                        Log.e("Error Timer",
                          e.getLocalizedMessage());
                    }
                }
            });
        }
    };
    timer.schedule(doVolleyRequest, 0, 1000); }
```

La función encapsula un conjunto de componentes (*handler*, *TimerTask* y *Runnable*) que permiten repetir una acción cada cierto periodo de tiempo. En esta función en concreto,

cada segundo. No confundir con la función del *LocationManager*, que se encarga sólo de recoger y crear un objeto JSON. Esta función tiene el cometido de enviar dicho JSON, siempre y cuando tengamos la validación de la bandera *start*, que recogimos al principio del servicio. Así, el servicio recoge los datos del dispositivo continuamente desde que es iniciado, pero no es hasta que *start* pasa a ser *true* cuando los datos se envían a la base de datos.

9.5.2.1.3 onDestroy

Aquí debemos liberar recursos como subprocesos, *listeners* y demás, ya que este método es llamado por el sistema cuando el servicio ya no se utiliza y va a ser destruido.

```
@Override
public void onDestroy() {
    if(locationManager != null){
        //noinspection MissingPermission
        locationManager.removeUpdates(listener);
    }
    if (timer != null) timer.cancel();
    start = false;
    super.onDestroy();
}
```

Si han sido creados, se cancelan las actualizaciones del *LocationManager* y el temporizador. Además devolvemos la bandera *start* a su estado por defecto de *false*.

9.5.2.1.4 stopService

No tiene más, para el servicio bajo petición de otro componente vía *intent*. Si el servicio no había sido lanzado y se llama a este método, nada ocurre.

```
@Override
public boolean stopService(Intent name) {
    return super.stopService(name);
}
```

9.5.2.1.5 Métodos Interfaz Volley

Puesto que las funciones que utilizamos de Volley en este caso son POST, es decir, escribimos nosotros la información en vez de leerla, las respuestas no tienen más utilidad que informar de que el mensaje ha sido transmitido correctamente.

```
@Override
public void myOnResponse(String requestType, JSONObject
  response) {

    switch (requestType) {
        case "gpsdata":
```

```
            Log.d(TAG, "Enviado JSON con datos GPS");
            break;

        case "token":
            Log.d(TAG, "Enviado JSON con token");
            break;
        default:
            Log.d(TAG, "requestType no soportado");
            break;
    }
}
```

9.5.2.2 Servicio Photo_Service

Android actualizó recientemente su API dedicada a la captura de imágenes. El paquete *camera2* reemplaza ya la obsoleta clase *Camera*, y con él vienen una gran cantidad de nuevas funciones y llamadas que permiten controlar y manipular las imágenes capturadas con un enfoque distinto.

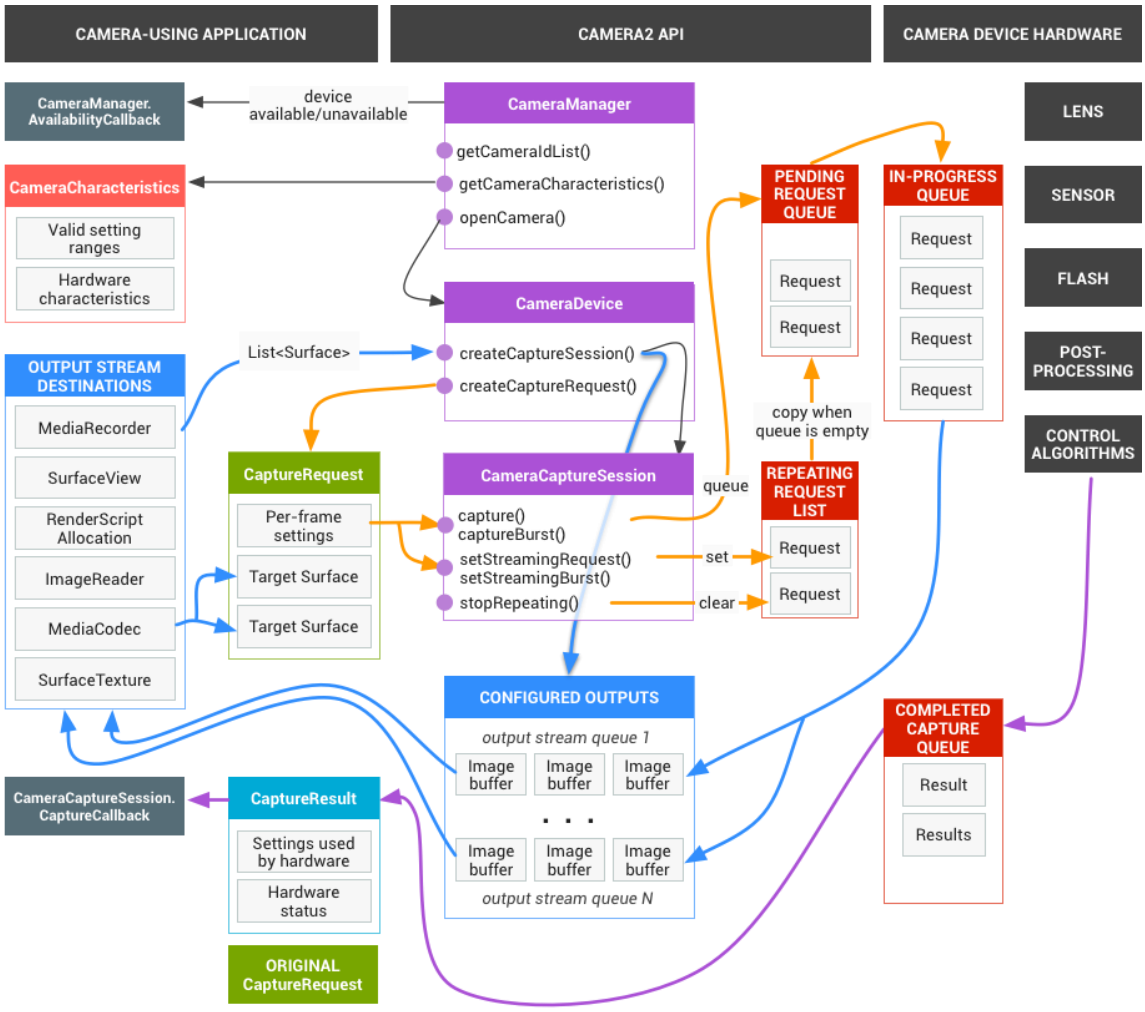

Figura 20: Diagrama de la estructura de la API camera2 de Google

En comparación con la API previa, *camera2* puede resultar bastante intimidante debido a su extensión. Para hacernos una idea, el ejemplo de uso del paquete dado por Android cuenta con 1036 líneas de código. Aún así, no debemos alarmarnos. Nuestro código no necesita todas y cada una de las funciones que ofrece *camera2*, por lo que no es tan extenso, de hecho no llega a las 300 líneas. Indistintamente, intentaremos simplificar al máximo nuestro código para facilitar su lectura y comprensión.

9.5.2.2.1 onStartCommand

Método llamado al iniciarse el servicio. En él recabaremos la bandera *start* proveniente de la notificación push, instanciaremos el *listener* necesario para la implementación de Volley y llamamos a la función *readyCamera()* que da comienzo a la captura de imágenes.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    volleyListener = this;
    start = intent.getExtras().getBoolean("start", false);
    readyCamera();
    return super.onStartCommand(intent, flags, startId);
}
```

9.5.2.2.2 readyCamera

El primer paso para capturar una imagen es instanciar un *CameraManager*. Este objeto nos permitirá gestionar todas las cámaras de nuestro dispositivo. Necesario ya que cada vez más el mercado tiende a incorporar al menos dos cámaras (frontal y trasera), y estas poseen distintas características.

```
public void readyCamera() {
    CameraManager manager = (CameraManager)
      getSystemService(CAMERA_SERVICE);

    try {
        String pickedCamera = getCamera(manager);
        manager.openCamera(pickedCamera, cameraStateCallback,
          null);
        imageReader = ImageReader.newInstance(854, 480,
          ImageFormat.JPEG, 2);
        imageReader.setOnImageAvailableListener(
          onImageAvailableListener, null);
        } catch (CameraAccessException e){
            Log.e(TAG, e.getMessage());
        }
    }
}
```

Para elegir qué cámara utilizamos llamamos a la función *getCamera*, una pequeña función desarrollada por nosotros, igual que *readyCamera*, que nos devuelve la identificación de la cámara que hemos elegido (en este caso la frontal) y almacena las características de la cámara en la variable global *characteristics*.

```
public String getCamera(CameraManager manager){
    try {
        for (String cameraId : manager.getCameraIdList()) {
            characteristics = manager.getCameraCharacteristics(
              cameraId);
            int cOrientation = characteristics.get(
              CameraCharacteristics.LENS_FACING);
            if (cOrientation == CAMERACHOICE) {
                mSensorOrientation = characteristics.get(
                  CameraCharacteristics.SENSOR_ORIENTATION);
                return cameraId;
            }
        }
    } catch (CameraAccessException e){
        e.printStackTrace();
    }
    return null;
}
```

Volviendo a *readyCamera*, seguimos abriendo la cámara seleccionada (método *openCamera* del objeto *manager*), y aquí vemos por primera vez, pasado como argumento, un *StateCallback*. Representan llamadas ejecutadas por el propio sistema conforme el proceso de captura de imágenes va pasando por cada una de sus distintas fases. En este caso el *StateCallback* responde al estado de la cámara elegida, más adelante lo veremos con más detalle.

Para finalizar, se instancia el *imageReader* y su *listener*. El primero es el objeto que contendrá la imagen en sí, y el segundo nos permite controlar el proceso de captura una vez la imagen esté lista para ser tomada.

9.5.2.2.3 CameraDevice.StateCallback

Como mencionábamos antes, objeto que controla las respuestas a las llamadas sobre el estado de la cámara.

```
protected CameraDevice.StateCallback cameraStateCallback = new
  CameraDevice.StateCallback() {
    @Override
    public void onOpened(@NonNull CameraDevice camera) {
        Log.d(TAG, "CameraDevice.StateCallback onOpened");
        cameraDevice = camera;
        actOnReadyCameraDevice();
    }
```

```
    @Override
    public void onDisconnected(@NonNull CameraDevice camera) {
        Log.w(TAG, "CameraDevice.StateCallback
          onDisconnected");
    }

    @Override
    public void onError(@NonNull CameraDevice camera, int error)
    {
      Log.e(TAG, "CameraDevice.StateCallback onError " + error);
    }
};
```

Básicamente lo utilizamos para llamar a la función *actOnReadyCameraDevice()*, la cual responde con el siguiente código:

```
public void actOnReadyCameraDevice(){
    try {
        cameraDevice.createCaptureSession(Arrays.asList(
         imageReader.getSurface()), sessionStateCallback, null);
    } catch (CameraAccessException e){
        Log.e(TAG, e.getMessage());
    }
}
```

Función sencilla, crea una sesión de capturas, necesaria siempre que queramos tomar una foto o reprocesar imágenes tomadas con anterioridad en la misma sesión. Si nos fijamos, incorpora su propio *StateCallback*, el cual veremos a continuación.

9.5.2.2.4 CameraCaptureSession.StateCallback

Objeto que nos permite seguir el proceso de una petición de captura enviado a la cámara.

En esta función se introduce la bandera *flag_session*. Ésta nos permite evitar errores de ejecución relacionados con la sesión. Se activa cuando la sesión es creada correctamente (*StateCallback onConfigured*), y se desactiva, si no lo estaba ya, en caso de que falle. Por otro lado, cuando todo está listo (*onReady StateCallback*) se vincula la sesión al servicio y si *flag_session* es *true* o lo que es lo mismo, existe sesión, se configura una serie de peticiones que se irán repitiendo ininterrumpidamente mediante *setRepeatingRequest*. Como es de esperar, estas peticiones son peticiones para realizar capturas de imagen. El primero de los argumentos que se le pasa a la función anterior es *createCaptureRequest()*.

```
protected CameraCaptureSession.StateCallback
  sessionStateCallback = new
  CameraCaptureSession.StateCallback() {
    @SuppressLint("NewApi")
    @Override
    public void onReady(CameraCaptureSession session) {
        Photo_Service.this.session = session;
```

```
        if (flag_session) {
            try {
                session.setRepeatingRequest(
                  createCaptureRequest(), null, null);
                cameraCaptureStartTime =
                  System.currentTimeMillis ();
            } catch (CameraAccessException e) {
                Log.e(TAG, e.getMessage());
            }
        } else { Log.d(TAG,"Session Flag: Closed");}
    }
    @Override
    public void onConfigured(CameraCaptureSession session) {
        flag_session = true;
    }
    @Override
    public void onConfigureFailed(@NonNull CameraCaptureSession
      session) {
        flag_session = false;
    }
};
```

9.5.2.2.5 createCaptureRequest

Función propia que encapsula el código necesario para construir una petición de una captura de imagen.

```
protected CaptureRequest createCaptureRequest() {
    try {
        CaptureRequest.Builder builder =
          cameraDevice.createCaptureRequest(
          CameraDevice.TEMPLATE_STILL_CAPTURE);
        builder.addTarget(imageReader.getSurface());
        builder.set(CaptureRequest.JPEG_ORIENTATION, 0);
        return builder.build();
    } catch (CameraAccessException e) {
        Log.e("Error@CaptureRequest", e.getMessage());
        return null;
    }
}
```

9.5.2.2.6 ImageReader.OnImageAvailableListener

Se trata de una interfaz que nos notifica cuando una nueva imagen está disponible.

```
protected ImageReader.OnImageAvailableListener
  onImageAvailableListener = new
  ImageReader.OnImageAvailableListener() {
    @Override
    public void onImageAvailable(ImageReader reader) {
        Image img = reader.acquireLatestImage();
        if (img != null) {
            if (System.currentTimeMillis () >
             cameraCaptureStartTime + CAMERA_CALIBRATION_DELAY){
                processImage(img);
            }
            img.close();
```

```
            }
        }
};
```

Cuando esto ocurre, se crea un objeto de tipo *Image* y se guarda en él la imagen proveniente del *ImageReader*. Acto seguido, si se cumple la restricción de tiempo, que básicamente es que hayan pasado 500 ms (*CAMERA_CALIBRATION_DELAY*), se procesa la imagen capturada en la variable *img*. Este periodo de tiempo se deja pasar con el objetivo de que el servicio capture suficientes imágenes para que se ajusten gradualmente los parámetros de exposición y así evitar que las capturas salgan oscuras, un problema recurrente en la API *camera2*.

9.5.2.2.7 processImage

Función propia encargada de recibir la imagen, procesarla y comprimirla en el formato Base64. Además de eso, si se cumple que es la primera vez que se entra al bucle, lo cual sabemos a la bandera *flag_first*, se llama a la función *setImageTimer()*. Ésta configura un temporizador para mandar la imagen mediante Volley cada 5 s. Es un temporizador idéntico al utilizado en el servicio *GPS_Service*, por lo que vamos a obviar su código.

```
private void processImage(Image image){
    ByteBuffer buffer;
    byte[] bytes;
    if(image.getFormat() == ImageFormat.JPEG) {
        buffer = image.getPlanes()[0].getBuffer();
        bytes = new byte[buffer.remaining()];
        buffer.get(bytes);
        imagb64 = Base64.encodeToString(bytes, Base64.DEFAULT);
        image.close();
        if (flag_first) {
            setImageTimer();
            flag_first = false;
        }
    }
}
```

9.5.2.2.8 onDestroy

Ya conocemos este método. Liberamos recursos en él.

```
@Override
public void onDestroy() {
    flag_first = false;
    flag_session = false;
    start = false;
    if (session != null){
        session.close();
    }
    if (timer != null){
```

```
        timer.cancel();
    }
}
```

Los métodos *stopService* junto con los de implementación de Volley son idénticos a los de *GPS_Service*, por lo que también los obviaremos.

9.5.2.3 Servicio FirebaseMessagingService

Servicio encargado de gestionar la llegada de las notificaciones push y crear su correspondiente notificación.

9.5.2.3.1 onMessageReceived

Método desarrollado por Android que es llamado cuando un mensaje es recibido.

```
@Override
public void onMessageReceived(RemoteMessage remoteMessage) {
    super.onMessageReceived(remoteMessage);
    if (remoteMessage.getNotification() != null) {
        mostrarNotificacion(
          remoteMessage.getNotification().getTitle(),
          remoteMessage.getNotification().getBody());
    }
    if (remoteMessage.getData().size() > 0) {
        String id = remoteMessage.getData().get("regatas_id");
            String start = remoteMessage.getData().get("start");
            HashMap pushData = new HashMap();
            pushData.put("regatas_id", id);
            pushData.put("start", start);
            showPushData(pushData, MainActivity.class);
    }
}
```

Se evalúan dos situaciones. En la primera se comprueba si el mensaje tiene notificación. En caso positivo se recaba el título y el cuerpo de ésta y se pasan como argumentos a la función de creación propia *mostrarNotificacion*.

```
private void mostrarNotificacion(String title, String body) {
    Intent intent = new Intent(this, MainActivity.class);
    intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
    PendingIntent pendingIntent = PendingIntent.getActivity(
      this, 0, intent, PendingIntent.FLAG_ONE_SHOT);
    NotificationCompat.Builder notificationBuilder = new
      NotificationCompat.Builder(
      getApplicationContext(), "regatas");
    notificationBuilder.setAutoCancel(true)
        .setDefaults(Notification.DEFAULT_ALL)
        .setWhen(System.currentTimeMillis())
        .setSmallIcon(R.drawable.ic_regata_not)
        .setPriority(Notification.PRIORITY_MAX)
        .setContentTitle(title)
        .setContentText(body)
        .setContentInfo("Info");
```

```
    NotificationManager notificationManager =
      (NotificationManager)
      getApplicationContext().getSystemService(Context.
      NOTIFICATION_SERVICE);
    notificationManager.notify(1, notificationBuilder.build());
}
```

Puede parecer una función densa, pero realmente lo único que hace es configurar los parámetros que se muestran en la notificación como pueden ser el icono que la representa, título y descripción, prioridad, hora a la que ha llegado, etc.

La segunda situación se da si la notificación, aparte de título y cuerpo, contiene metadatos. En este caso, se recogen y se muestran en la actividad principal mediante la función *showPushData*, creada por nosotros:

```
private void showPushData(HashMap data, Class clase){
    Intent i = new Intent(getApplicationContext(), clase);
    i.putExtra("data", data);
    i.setAction("START_SIGNAL");
    LocalBroadcastManager.getInstance(
      getApplicationContext()).sendBroadcast(i);
}
```

Sencilla, la función recoge los datos que se quieren mandar, la clase a la que se quiere mandar y se transmite mediante un *intent*.

9.5.2.4 Servicio FirebaseInstanceIDService

El servicio menos extenso de la aplicación, pero no por ello menos importante. Se encarga de obtener los tokens de la nube de Firebase (FCM) y mandarlos a la base de datos mediante Volley.

9.5.2.4.1 onTokenRefresh

Método llamado cuando los tokens deben ser refrescados, por lo que no debería ser llamado muy frecuentemente.

```
@Override
public void onTokenRefresh() {
    super.onTokenRefresh();
    String token = FirebaseInstanceId.getInstance().getToken();
    enviarTokenAlServidor(token);
}
```

Se recoge el token desde el servicio de Firebase y se llama a la función *enviarTokenAlServidor* pasándole la variable que lo contiene:

```
private void enviarTokenAlServidor(String token) {
```

```
    jsonObject = new JSONObject();
    try {
        jsonObject.put("token", token);
    } catch (JSONException e) {
        e.printStackTrace();
    }
    requestType = "token";
    volleyUtils = new VolleyUtils(getApplicationContext(),
      volleyListener, requestType, jsonObject,
      getResources().getString(R.string.token),
      Request.Method.POST);
    volleyUtils.runJSONObjRequest();
}
```

Crea un objeto JSON con el token y su identificación para mandarlo mediante nuestra interfaz Volley. Los métodos para implementar dicha interfaz son los mismos de siempre, por los que es redundante volver a ponerlos en el anexo.

9.5.3 Manifiesto

Archivo que sirve para identificar los componentes, estructura y características de la aplicación.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="sfc.tfg.com.gpsvolleylocationfinal">

    <uses-permission android:name =
      "android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name =
      "android.permission.INTERNET" />
    <uses-permission android:name="android.permission.CAMERA"/>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_applogo"
        android:label="@string/app_name"
        android:screenOrientation="portrait"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name =
                  "android.intent.action.MAIN" />
                <category android:name =
                  "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".GPS_Service"/>
        <service android:name=".Photo_Service"/>
        <service
            android:name=".FirebaseMessagingService">
            <intent-filter>
                <action android:name =
                  "com.google.firebase.MESSAGING_EVENT"/>
            </intent-filter>
```

```
        </service>
        <service
            android:name=".FirebaseInstanceIDService">
            <intent-filter>
                <action android:name =
                  "com.google.firebase.INSTANCE_ID_EVENT"/>
            </intent-filter>
        </service>
    </application>
</manifest>
```

Al igual que en los códigos de los *layouts*, están escritos en XML y son más intuitivos a la hora de leerlos que los escritos en Java. Con una simple ojeada podemos ver los permisos que utiliza la aplicación (localización y acceso a la red), el nombre de la aplicación, su logo, las actividades y servicios que la componen.

9.6 Anexo VI: Aplicación App Regatas

Anexo dedicado a la aplicación App Regatas. En éste desgranaremos las funciones que contiene a nivel de código y los procesos necesarios para poder construir la app.

9.6.1 Google Maps API

Para poder utilizar las funciones de los mapas de Google es necesario seguir un proceso de configuración que sirve para identificar la aplicación de cara a la compañía. Este servicio no es gratuito en sí, pero Google permite utilizarlo durante los primeros 12 meses sin cargo alguno (hasta un límite de 300 $). Una vez acabado este periodo, existen varias tarifas que se pueden ajustar a la aplicación en función del uso de la API.

El proceso de configuración no tiene muchos pasos, solo puede alargarse si no contamos con una cuenta de Google o ésta no tiene asociada una cuenta de facturación (sólo necesaria para verificar que no eres un robot).

1. **Conseguir una clave de la API:**

Clicando en el botón "GET STARTED" que aparece en la página de la plataforma de mapas de Google (https://developers.google.com/maps/documentation/android-sdk/signup) podremos elegir los servicios en concreto que necesitamos para nuestra aplicación. Una vez elegido/s, tendremos que seleccionar el proyecto de Android en el que queremos implementarlo y escoger o crear la cuenta de facturación. Una vez completados los pasos, se nos dará la clave y se nos instará a restringir su uso.

2. **Añadir la clave de la API a nuestra aplicación:**

Dentro de Android Studio, tendremos que añadir el siguiente código en *AndroidManifest.xml*, dentro de la etiqueta <*application*>:

```
    <meta-data
        android:name="com.google.android.geo.API_KEY"
        android:value="TU_CLAVE_API"/>
```

Donde deberemos cambiar el campo *value* de "TU_CLAVE_API" por el valor de la clave obtenida en el primer paso. Por último, para hacer efectiva la configuración, tenemos que guardar *AndroidManifest.xml* y ejecutar la reconstrucción de la aplicación.

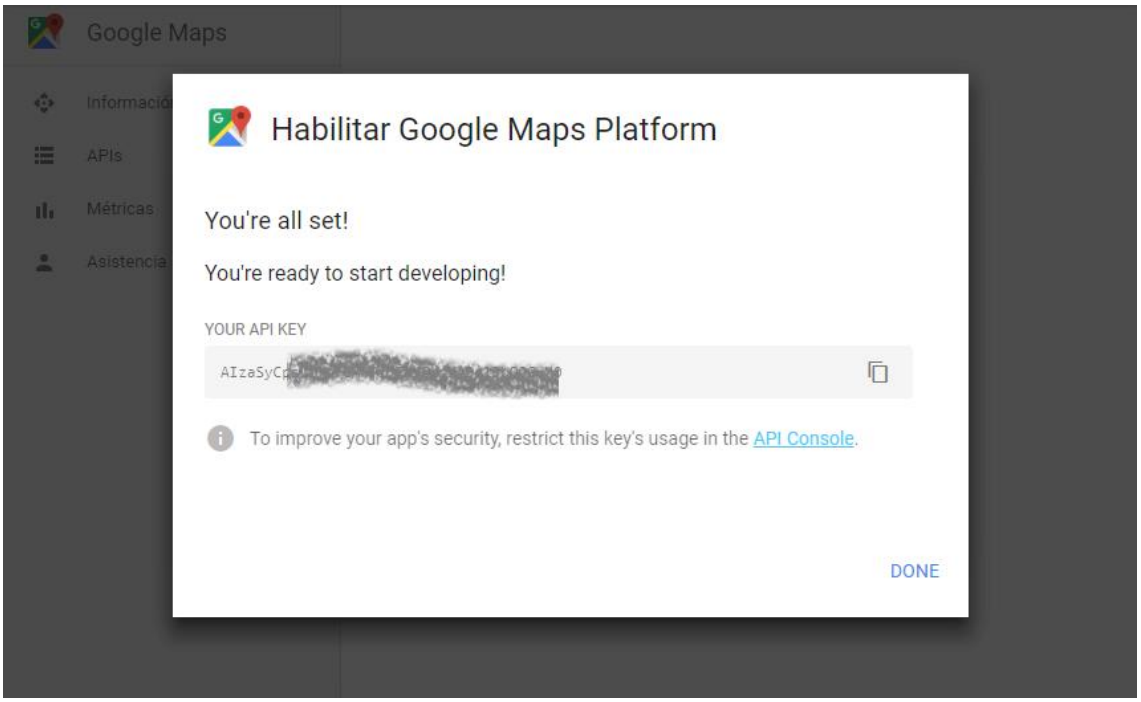

Figura 21: Clave para habilitar el uso de la API de Google Maps en nuestro proyecto de Android Studio

9.6.2 Actividad principal

Clase que hará básicamente de contenedor de los fragments. También implementa el menú lateral que permite cambiar de un fragment a otro.

9.6.2.1 Layout

Al implementar el panel lateral, el diseño correspondiente a la actividad principal es un poco más complejo de lo normal. Aquí veremos el código de los archivos necesarios para componer dicho panel:

- **activity_main.xml:**

  Diseño de la actividad principal. En él se instancian elementos auxiliares, los cuales se verán a continuación.

```
<android.support.v4.widget.DrawerLayout
xmlns:android "http://schemas.android.com/apk/res/android
xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
```

```
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">
    <include
        layout="@layout/app_bar_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main"
        app:menu="@menu/menu_lateral" />
</android.support.v4.widget.DrawerLayout>
```

- **app_bar_main.xml:**

  Sirve para definir la barra superior que se muestra en la aplicación y en la que que más tarde, mediante código Java, instanciaremos el botón que abre el menú lateral.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android=http://schemas.android.com/apk/res/android
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="sfc.tfg.com.appregatasfinal.MainActivity">
    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />
    </android.support.design.widget.AppBarLayout>
    <include layout="@layout/content_main" />
</android.support.design.widget.CoordinatorLayout>
```

- **menu_lateral.xml:**

  Este archivo es el que da forma a los apartados que pueblan el menú lateral.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android =
    "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:showIn="navigation_view">
    <group android:checkableBehavior="single">
        <item
```

```
            android:id="@+id/recorridos"
            android:icon="@drawable/ic_recorrido"
            android:title="Recorrido" />
        <item
            android:id="@+id/seguimiento"
            android:icon="@drawable/ic_seguimiento"
            android:title="Seguimiento" />
        <item
            android:id="@+id/camara"
            android:icon="@drawable/ic_camara"
            android:title="Directo" />
        <item
            android:id="@+id/juez"
            android:icon="@drawable/ic_juez"
            android:title="Juez" />
    </group>
</menu>
```

- **nav_header_main.xml:**

  Contiene las características de la cabecera del menú lateral. Púramente estético.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="@dimen/nav_header_height"
    android:background="@drawable/side_nav_bar"
    android:gravity="bottom"
    android:orientation="vertical"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:theme="@style/ThemeOverlay.AppCompat.Dark">
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="78dp"
        android:layout_height="69dp"
        android:adjustViewBounds="false"
        android:cropToPadding="false"
        android:paddingTop =
          "@dimen/nav_header_vertical_spacing"
        app:srcCompat="@mipmap/ic_applogo" />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop =
          "@dimen/nav_header_vertical_spacing"
        android:text="App Regatas"
        android:textAppearance =
          "@style/TextAppearance.AppCompat.Body1" />
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un TFG de Saúl Fernández Candela" />
```

```
</LinearLayout>
```

9.6.2.2 onCreate

En él declaramos los elementos que definen la actividad principal.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    drawer = findViewById(R.id.drawer_layout);
    ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(
      this, drawer, toolbar, R.string.navigation_drawer_open,
      R.string.navigation_drawer_close);
    drawer.addDrawerListener(toggle);
    toggle.syncState();

    NavigationView navigationView = findViewById(R.id.nav_view);
    navigationView.setNavigationItemSelectedListener(this);
    textView = findViewById(R.id.textView);
}
```

En el primer párrafo se configura el diseño de la actividad haciendo referencia al archivo *activity_main.xml* y también se instancia la barra de herramientas, dónde aparecerá el botón que sirve para abrir el menú del panel lateral (aparte de deslizar el dedo desde el borde izquierdo de la pantalla). En el segundo párrafo se declara el panel lateral o cajón y se le vincula a éste. Por último, se instancia el contenedor de texto que da la bienvenida y un *navigationView*, objeto que nos permite crear un *listener* para controlar la aplicación en función de los ítems seleccionados en el panel lateral.

9.6.2.3 onBackPressed

Método llamado cuando el usuario pulsa el botón de retroceder en el dispositivo.

```
@Override
public void onBackPressed() {
    if (drawer.isDrawerOpen(GravityCompat.START)) {
        drawer.closeDrawer(GravityCompat.START);
    } else super.onBackPressed();
}
```

Comprueba si el panel está abierto, y en caso afirmativo lo cierra. En caso contrario no ocurre nada.

9.6.2.4 onNavigationItemSelected

Método que es declarar para implementar la arquitectura de fragments.

```
@SuppressWarnings("StatementWithEmptyBody")
@Override
public boolean onNavigationItemSelected(MenuItem item) {
    int id = item.getItemId();

    if (id == R.id.recorridos) {
        this.miFragment = new RecorridosFragment();
    } else if (id == R.id.seguimiento) {
        this.miFragment = new TrackerFragment();
    } else if (id == R.id.juez) {
        this.miFragment = new JuezFragment();
    } else if (id == R.id.camara) {
        this.miFragment = new CameraFragment();
    }

    fragmentManager =  getSupportFragmentManager();
    android.support.v4.app.FragmentTransaction
      fragmentTransaction =
      fragmentManager.beginTransaction();
    fragmentTransaction.replace(
      R.id.contenedor_fragments, this.miFragment);
    fragmentTransaction.commit();
    textView.setVisibility(View.GONE);
    drawer.closeDrawer(GravityCompat.START);
    return true;
}
```

El método recibe un objeto de tipo i*tem*, con el cual obtenemos la identificación de qué elemento ha sido seleccionado en el menú del panel lateral y lo almacenamos en la variable *id*. Ahora, dependiendo de qué identificación recibamos, se instancia su fragment correspondiente. Acto seguido se hace desaparecer el texto de bienvenida, se cargar el fragmento instanciado y se cierra el panel lateral, que se encontraba abierto.

9.6.3 Fragments

Como su propio nombre indica, son fragmentos de una actividad superior. Representan los elementos básicos de la estructura de la aplicación, donde cada uno desempeña una función distinta.

9.6.3.1 Fragmento Recorridos

Muestra en el mapa la trayectoria completa del cliente durante el transcurso de la regata.

9.6.3.1.1 Layout

Aquí veremos el fichero que describe la apariencia de este fragmento.

```
<FrameLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context =
      "sfc.tfg.com.appregatasfinal.RecorridosFragment">

    <com.google.android.gms.maps.MapView
        android:id="@+id/mapViewRecorridos"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</FrameLayout>
```

Simplemente, incluye un contenedor de mapa o *MapView*.

9.6.3.1.2 onCreateView

Método propio de los fragments, equivalente al *onCreate* de las actividades.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup
  container, Bundle savedInstanceState) {
    View rootView = inflater.inflate(
      R.layout.fragment_recorridos, container, false);
    mMapView = rootView.findViewById(R.id.mapViewRecorridos);
    mMapView.onCreate(savedInstanceState);
    volleyListener = this;
    mMapView.onResume();

    try {
        MapsInitializer.initialize(
          getActivity().getApplicationContext());
    } catch (Exception e) {
        e.printStackTrace();
    }

    mMapView.getMapAsync(new OnMapReadyCallback() {

        @Override
        public void onMapReady(GoogleMap mMap) {
            googleMap = mMap;
            jArray = new JSONArray();
            volleyUtils = new VolleyUtils(
              getContext(), volleyListener, requestType, jArray,
              getResources().getString(R.string.pos),
              Request.Method.GET);
            volleyUtils.runJsonArrayRequest();
        }
    });
    return rootView;
}
```

Inicializamos los componentes que conforman el fragment: su *layout*, el *listener* de Volley y el mapa. Una vez que el mapa está listo procedemos mediante la interfaz de Volley a acceder a nuestra API que nos provee las posiciones.

9.6.3.1.3 Métodos Interfaz Volley

En este caso debemos trabajar con los datos que recibamos en la respuesta de la petición de Volley, ya que estamos pidiendo datos en vez de insertándolos.

```
@Override
public void myOnResponse(String requestType, JSONArray response) {
    LatLng pos = null;
    try {
        jArray = response;
        for (int i = 0; i < jArray.length(); i++) {
            jsonObject = jArray.getJSONObject(i);
            pos = new LatLng(jsonObject.getDouble("latitud"),
              jsonObject.getDouble("longitud"));
            MarkerOptions marker =
              new MarkerOptions().position(pos);
            if ((i >= 1) && (i < jArray.length() - 1)) {
                marker.icon(
                  BitmapDescriptorFactory.fromResource(
                  R.drawable.ic_dot));
                if (i % 7.5 == 0) googleMap.addMarker(marker);
            } else {
                googleMap.addMarker(
                  new MarkerOptions().position(pos));
            }
        }
        googleMap.animateCamera(
          CameraUpdateFactory.newLatLngZoom(pos, 17));
    } catch (JSONException e) {
        Log.e("ERROR on myOnResponse", e.getLocalizedMessage());
    }
}

@Override
public void myOnResponse(String requestType, JSONObject
  response) {

    }

@Override
public void myOnError(String message) {
    Log.e("ERROR:", message);
    Toast.makeText(getContext(),"Error al conectar con el
      servidor", Toast.LENGTH_SHORT).show();
}
```

El objeto que recibimos como respuesta a la llamada a la API es un array de JSONs, por lo que debemos procesarlo para presentarlo al usuario mediante un bucle *for*. Dentro de él haremos dos distinciones, una anidada dentro de la otra. La primera nos permite concretar si nos encontramos ante la primera o última posición recogida, o bien entre las de en medio, ya que para una visualización más clara pondremos un marcador distinto a las posiciones intermedias que a las del principio y final. El segundo *if*, definido dentro del primero, actúa como divisor. El array es muy extenso debido a la precisión del GPS,

por lo que si pusiéramos un marcador por cada elemento obtendríamos una línea negra opaca pintada encima del mapa, poco atractiva de cara al cliente. Por este motivo solo pintamos una porción de las posiciones.

A continuación, tenemos sobrescrito el método de respuesta para un objeto JSON también pero vacío, debido a que en este fragmento no es necesario, pero en otros lo será. Y por último, tenemos el método de la interfaz de Volley que es llamado cuando la respuesta da error y utilizamos para comunicarlo tanto al usuario como al desarrollador para su depuración.

9.6.3.2 Fragmento Seguimiento

En este fragmento se muestra únicamente la última posición registrada. Está pensada para ser utilizada mientras la regata o el entrenamiento está teniendo lugar, por lo que se actualiza automáticamente cada segundo.

9.6.3.2.1 Layout

Archivo que describe la apariencia del fragmento.

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="sfc.tfg.com.appregatasfinal.TrackerFragment"
    android:id="@+id/tracker_fragment">

    <com.google.android.gms.maps.MapView
        android:id="@+id/mapViewTracker"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <ImageButton
        android:id="@+id/btnRefresh"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="right|bottom"
        android:background="@drawable/round_shape"
        android:contentDescription="Refresh Button"
        android:src="@drawable/ic_refresh"
        android:layout_margin="20dp"/>
</FrameLayout>
```

Incluye un contenedor de mapa y un botón con imagen (*ImageButton*).

9.6.3.2.2 onCreateView

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup
  container, Bundle savedInstanceState) {
    View rootView = inflater.inflate(
      R.layout.fragment_tracker, container, false);
    mMapView = rootView.findViewById(R.id.mapViewTracker);
    mMapView.onCreate(savedInstanceState);
    mMapView.onResume();

    ImageButton btn_Refresh =
      rootView.findViewById(R.id.btnRefresh);
    volleyListener = this;
    requestType = "getJsonObject";

    try {
        MapsInitializer.initialize(
          getActivity().getApplicationContext());
    } catch (Exception e) {
            e.printStackTrace();
    }

    mMapView.getMapAsync(new OnMapReadyCallback() {
        @Override
        public void onMapReady(GoogleMap mMap) {
            googleMapTracker = mMap;
            callVolleyRequest();
        }
    });


    btn_Refresh.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            volleyUtils = new VolleyUtils(
              getContext(), volleyListener, requestType,
              jsonObject,
              getResources().getString(R.string.lastpos),
              Request.Method.GET);
            volleyUtils.runJsonObjectRequest();
        }
  });
    return rootView;
}
```

Código similar al del fragmento anterior pero con algunas líneas adicionales. Éste incorpora un botón que sirve para llamar a la API que provee la última posición, por si la espera de la llamada automática cada segundo no es lo suficiente rápida para el usuario, o se quiere comprobar si el servicio está funcionando. También, por defecto el fragmento llamará a la función *callVolleyRequest()* conforme el mapa esté listo para poder interactuar con él.

```
public void callVolleyRequest() {
    final Handler handler = new Handler();
    timer = new Timer();
    doVolleyRequest = new TimerTask() {
        @Override
        public void run() {
            handler.post(new Runnable() {
                public void run() {
                    try {
                        volleyUtils = new
                          VolleyUtils(getContext(),
                          volleyListener, requestType,
                          jsonObject, getResources().
                          getString(R.string.lastpos),
                          Request.Method.GET);
                        volleyUtils.runJsonObjectRequest();
                    } catch (Exception e) {
                        Log.e("Error Timer",
                          e.getLocalizedMessage());
                    }
                }
            });
        }
    };
    timer.schedule(doVolleyRequest, 0, 1000);
}
```

Esta función de creación propia es un temporizador, igual que los que se encuentran en la aplicación GPS Volley Location. Recoge todos los componentes necesarios para repetir una acción cada cierto periodo de tiempo en Android, en este caso, a cada segundo.

9.6.3.2.3 Métodos Interfaz Volley

Igual que en el fragmento anterior utilizábamos sólo el método de respuesta de Volley para arrays JSON y dejábamos vacío el correspondiente a un único objeto, aquí ocurre lo contrario.

```
@Override
public void myOnResponse(String requestType, JSONObject
  response) {
    if (isAdded())
        try {
            jsonObject = response;
            pos = new LatLng(jsonObject.getDouble("latitud"),
              jsonObject.getDouble("longitud"));
            double rumbo = jsonObject.getDouble("rumbo");
            double vel = jsonObject.getDouble("velocidad");

            if (persistentMarker == null) {
                markerOptions = new
                  MarkerOptions().position(pos);
                markerOptions.icon(
                  BitmapDescriptorFactory.fromResource(
                  R.drawable.ic_sailboat2));
                markerOptions.anchor(0.5f, 0.5f);
                markerOptions.flat(true);
```

```
                markerOptions.rotation((float) rumbo);
                markerOptions.title("Velocidad: "+vel+" m/s");
                persistentMarker =
                  googleMapTracker.addMarker(markerOptions);
                googleMapTracker.animateCamera(
                  CameraUpdateFactory.newLatLngZoom(pos, 17));
            }

            else {
                persistentMarker.remove();
                markerOptions.rotation((float) rumbo);
                markerOptions.title("Velocidad: "+vel+" m/s");
                persistentMarker =
                  googleMapTracker.addMarker(markerOptions);
            }

        } catch (JSONException e) {
            Log.e("Error onPostExecute",
              e.getLocalizedMessage());
            Toast.makeText(getContext(), "Error en
              onPostExecute", Toast.LENGTH_SHORT).show();
        }
    else Log.e("Error String", "String vacío");
}

@Override
public void myOnResponse(String requestType, JSONArray response) {
}
@Override
public void myOnError(String message) {
    Log.e("ERROR:", message);
}
```

Centrándonos el método *myOnResponse* para objetos JSON, podemos ver que manejar la respuesta puede tener más de lo que en un principio puede parecer. Esto se debe a que cada vez que se recibe una posición, debemos diferenciar entre si es la primera vez que vamos a situar un marcador en el mapa, o ya había otro. Si no hacemos esto, en vez de un único marcador que va cambiando de posición por el mapa a cada segundo, lo que obtendríamos sería una hilera de marcadores. Para evitar esto, lo que hacemos cada vez que se llama este método es evaluar si el marcador ha sido inicializado, es decir, si es nulo o no. En el caso de que lo sea lo configuramos mediante *markerOptions*, dándole su icono concreto y demás propiedades comunes, aparte de la posición, rumbo y velocidad. En caso contrario, lo que hacemos es borrar el marcador anterior, y crear otro nuevo con la configuración que ya se hizo en la primera iteración, conservada en *markerOptions*, pero actualizando los datos con los de la nueva posición.

9.6.3.3 Fragmento Cámara

La función de este fragment se reduce a mostrar la imagen enviada desde GPS Volley Location. O bien si el servicio no está retransmitiendo imágenes, mostrar un texto de "Servicio Offline".

9.6.3.3.1 Layout

Define la interfaz gráfica del fragmento. Incluye un contenedor de imagen y otro de texto.

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/camera_fragment">

    <ImageView
        android:id="@+id/iv_camera"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity=
          "center_vertical|center_horizontal"
        android:visibility="visible" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity =
          "center_vertical|center_horizontal"
        android:textSize="24sp"
        android:visibility="gone" />
</FrameLayout>
```

9.6.3.3.2 onCreateView

Al tener tan pocos elementos y una función simple, no contiene mucho código.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup
  container, Bundle savedInstanceState) {
    View rootView = inflater.inflate(R.layout.fragment_camera,
      container, false);
    ivCamera = rootView.findViewById(R.id.iv_camera);
    textView = rootView.findViewById(R.id.textView);
    jArray = new JSONArray();
    jsonObject = new JSONObject();
    volleyListener = this;
    requestType = "getJsonObject";
    callVolleyRequest();
    return rootView;
}
```

Se instancian el contenedor de texto y el de imagen, los objetos necesarios para realizar la petición de Volley y se llama a la función *callVolleyRequest*, encargada de configurar el temporizador que consulta la imagen cada cinco segundos. No mostraremos el código de esta función ya que es exactamente igual al del temporizador anterior, cambiando el tiempo de 1 segundo por 5.

9.6.3.3.3 Métodos Interfaz Volley

Como las peticiones en este fragmento son de solo una imagen, el método que utilizamos es el de un objeto JSON. Es el código que se ejecutará cada 5 segundos en respuesta a la petición del temporizador.

```
@Override
public void myOnResponse(String requestType, JSONArray response) {

}

@Override
public void myOnResponse(String requestType, JSONObject response) {
    try {
        encodedImage = response.getString("imagb64");
        if (!encodedImage.equals("stop")) {
            textView.setVisibility(View.GONE);
            ivCamera.setVisibility(View.VISIBLE);
            byte[] decodedString = Base64.decode(encodedImage,
              Base64.DEFAULT);
            Bitmap decodedByte = BitmapFactory.decodeByteArray(
              decodedString, 0, decodedString.length);
            Matrix matrix = new Matrix();
            matrix.postRotate(270);
            Bitmap scaledBitmap = Bitmap.createScaledBitmap(
              decodedByte, 854, 480, true);
            Bitmap rotatedBitmap = Bitmap.createBitmap(
              scaledBitmap, 0, 0, scaledBitmap.getWidth(),
              scaledBitmap.getHeight(), matrix, true);
            ivCamera.setImageBitmap(rotatedBitmap);
        } else {
            ivCamera.setVisibility(View.GONE);
            textView.setVisibility(View.VISIBLE);
            textView.setText("Servicio Offline");
        }
    } catch (JSONException e) {
        e.printStackTrace();
    }
}

@Override
public void myOnError(String message) {
    Log.e("myOnError", message);
}
```

Lo primero que hacemos al recibir la imagen es comprobar si el servicio está online u offline. Para ello lo que hacemos es ver si la respuesta contiene la cadena de caracteres

"stop". Si no se da esto, el servicio está funcionando y en consecuencia ocultaremos el *textView* y haremos visible el *imageView*. Después decodificaremos la imagen y la rotaremos y escalaremos para mostrarla con la orientación y relación de aspecto correcta en el contenedor de imagen. Por otro lado, si el texto sí es "stop" significa que el servicio se paró y como señal se mandó esta cadena a la base de datos, y lo que haremos será ocultar el *imageView* y hacer visible el *textView* con el texto "Servicio Offline".

9.6.3.4 Fragmento Juez

Utilizado para mandar las notificaciones push. Consta con un *EditText*, que permite al usuario introducir el código de la regata, y con dos botones. Uno que mandará una notificación que da comienzo a la regata y con ella el valor que permite, que GPS Volley Location comience a enviar datos, y otro botón que la finaliza y anula el envío de datos.

9.6.3.4.1 Layout

Interfaz gráfica que representa al fragmento.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android =
    "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/juez_fragment">

    <EditText
        android:id="@+id/editText"
        android:layout_width="200dp"
        android:layout_height="45dp"
        android:layout_marginBottom="32dp"
        android:contentDescription="Código Regata"
        android:ems="10"
        android:hint="Código Regata"
        android:inputType="number"
        android:textAlignment="textStart"
        app:layout_constraintBottom_toTopOf="@+id/inicio_btn"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent" />

    <Button
        android:id="@+id/inicio_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Inicio"
        app:layout_constraintBottom_toTopOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="parent" />
```

```
    <Button
        android:id="@+id/final_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:text="Final"
        app:layout_constraintTop_toBottomOf="@+id/inicio_btn"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent" />

</android.support.constraint.ConstraintLayout>
```

Incluye un contenedor de texto que insta a introducir los datos junto a al *editText* necesario para hacerlo y dos botones.

9.6.3.4.2 onCreateView

Igual que en el fragmento anterior, código muy escueto.

```
@Override
public View onCreateView(LayoutInflater inflater,
  ViewGroup container, Bundle savedInstanceState) {
    final View rootView = inflater.inflate(
      R.layout.fragment_juez, container, false);
      volleyListener = this;
      editText = rootView.findViewById(R.id.editText);
      btnInicio = rootView.findViewById(R.id.inicio_btn);
      btnFinal = rootView.findViewById(R.id.final_btn);
      btnInicio.setOnClickListener(this);
      btnFinal.setOnClickListener(this);
      return rootView;
}
```

Se instancian los elementos de interacción con el usuario, y se les asigna sus *listeners*, que esta vez se encuentran implementados en la clase, de ahí que se instancien pasándoles el puntero *this*.

9.6.3.4.3 onClick

Método para implementar las acciones de los botones.

```
@Override
public void onClick(View v){
    switch (v.getId()) {
        case R.id.inicio_btn:
            jsonObject = new JSONObject();
            try {
                jsonObject.accumulate("regatas_id",
                  editText.getText());
                jsonObject.accumulate("start", 1);
            } catch (JSONException e) {
                e.printStackTrace();
            }
            requestType = "push";
```

```
            volleyUtils = new VolleyUtils(getContext(),
              volleyListener, requestType, jsonObject,
              getResources().getString(
              R.string.fcm), Request.Method.POST);
            volleyUtils.runJsonObjectRequest();
            break;

        case R.id.final_btn:
            jsonObject = new JSONObject();
            try {
                jsonObject.accumulate("regatas_id",
                  editText.getText());
                jsonObject.accumulate("start", 0);
            } catch (JSONException e) {
                e.printStackTrace();
            }
            requestType = "push";
            volleyUtils = new VolleyUtils(getContext(),
              volleyListener, requestType, jsonObject,
              getResources().getString(
              R.string.fcm), Request.Method.POST);
            volleyUtils.runJsonObjectRequest();
            break;

        default:
            Log.e("Error en switch", "Caso no contemplado");
    }
}
```

Puesto que los dos botones se implementan mediante el puntero *this*, necesitamos una estructura *switch/case* para controlar las dos acciones que disparan cada uno. Básicamente, los dos contienen las mismas acciones, crean un objeto JSON, recogen el código de la regata introducido en el *editText* y lo introducen en el JSON, y lo envían a la misma API. Lo único en que se diferencian es el valor que se introduce en el parámetro "start" del JSON: 1 para la notificación del inicio, y 0 para la notificación del final.

En cuanto al código de implementación de la interfaz de Volley en este fragmento, lo obviamos ya que lo único que hacemos es confirmar el envío de la notificación, o comunicar el error para su depuración.

9.6.4 Manifiesto

La ficha técnica de la aplicación.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android"
    package="sfc.tfg.com.appregatasfinal">

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

```
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_applogo"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name =
                  "android.intent.action.MAIN" />
                <category android:name=
                     "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    <meta-data
            android:name="com.google.android.geo.API_KEY"
            android:value="@string/google_maps_key" />
    </application>

</manifest>
```

No muy extenso, de hecho más corto que el de GPS Volley Location. Esto se debe a que los fragmentos no se declaran. Como novedad podemos ver el apartado de “meta-data”, necesario para la implementación de la API de Google Maps.

10 BIBLIOGRAFÍA/WEBGRAFÍA

10.1 Libros

Stauffer, M., (2016), Laravel Up and Running: A framework for building modern PHP apps, O'Reilly Media, Inc.

10.2 Fuentes on-line

10.2.1 Laravel

Laravel. (2018). *Installation*. Recuperado de https://laravel.com/docs/5.5

Peña, Andres. [andrewsxx1]. (2017, septiembre 14). *Tutorial Laravel Restful api – Basico*. Recuperado de https://www.youtube.com/watch?v=DLRuHZqnw-4&t

10.2.2 Google Maps

Android Trainee. (2018). *Adding multiple marker locations in Google Maps Android API V2 and save it in Shared Preferences*. Recuperado de http://www.androidtrainee.com/adding-multiple-marker-locations-in-google-maps-android-api-v2-and-save-it-in-shared-preferences/

Codelabs Google. (2018). *Real-time Asset Tracking*. Recuperado de https://codelabs.developers.google.com/codelabs/realtime-asset-tracking/index.html?index=..%2F..%2Findex#0

Vujovic, F. [Filip Vujovic]. (2017, junio 17). *Android - Get GPS location via service*. Recuperado de https://www.youtube.com/watch?v=lvcGh2ZgHeA&t

Vujovic, F. [Filip Vujovic]. (2015, octubre 12). *How To Get GPS Location In Android*. Recuperado de https://www.youtube.com/watch?v=lvcGh2ZgHeA&t

Stack Overflow. (2015, mayo 15). *How to draw path as I move starting from my current location using Google Maps*. Recuperado de https://stackoverflow.com/questions/30249920/how-to-draw-path-as-i-move-starting-from-my-current-location-using-google-maps

Stack Overflow. (2013, octubre 14). *How to put Google Maps V2 on a Fragment using ViewPager*. Recuperado de https://stackoverflow.com/questions/19353255/how-to-put-google-maps-v2-on-a-fragment-using-viewpager

Google Maps Platform. (2018). *Creating a Map: Map Objects*. Recuperado de https://developers.google.com/maps/documentation/android-api/map?hl=es-419

10.2.3 Recabar JSON desde una URL

Victor Fontanos, C. (2018). *Android Development – Parsing JSON Data from a URL*. Recuperado de http://carlofontanos.com/android-development-parsing-json-data-from-a-url/

10.2.4 Repetir una tarea periódicamente

Stack Overflow. (2011, junio 30). *How to execute Async task repeatedly after fixed time intervals*. Recuperado de https://stackoverflow.com/questions/6531950/how-to-execute-async-task-repeatedly-after-fixed-time-intervals

10.2.5 Volley

Chakraborty, Arnab. (2018). *Asynchronous HTTP Requests in Android Using Volley*. Recuperado de http://arnab.ch/blog/2013/08/asynchronous-http-requests-in-android-using-volley/

Coding in Flow. [Coding in Flow]. (2017, noviembre 18). *How to Parse a Json Using Volley – SIMPLE GET REQUEST – Android Studio Tutorial*. Recuperado de https://www.youtube.com/watch?v=y2xtLqP8dSQ

Gupta, Aman. (2016, enero 15). *Return response data from function "onResponse" in Volley Library*. Recuperado de https://mobikul.com/how-to-send-json-post-request-using-volley-rest-api/

Stack Overflow. (2016, febrero 25). *How to make separate class for volley library and call all method of volley from another activity and get response?*. Recuperado de https://stackoverflow.com/questions/35628142/how-to-make-separate-class-for-volley-library-and-call-all-method-of-volley-from

Stack Overflow. (2015, noviembre 5). *How to create a proper Volley Listener for cross class Volley method calling*. Recuperado de https://stackoverflow.com/questions/33535435/how-to-create-a-proper-volley-listener-for-cross-class-volley-method-calling

10.2.6 De/codificar imágenes Base64

Mishra, Neeraj. (2018). *Android Convert Image to Base64 String or Base64 String to Image*. Recuperado de https://www.thecrazyprogrammer.com/2016/10/android-convert-image-base64-string-base64-string-image.html

Stack Overflow. (2014, diciembre 11). *Android resizing large image recieved as base64 string*. Recuperado de https://stackoverflow.com/questions/27416618/android-resizing-large-image-recieved-as-base64-string

10.2.7 Notificaciones Push

Brosy, Nicolas. (2017, noviembre 30). *Laravel-FCM. Recuperado de* https://github.com/brozot/Laravel-FCM

Khan, Belal. (2016, noviembre 5). *Firebase Cloud Messaging for Android using PHP and MySQL*. Recuperado de https://www.simplifiedcoding.net/firebase-cloud-messaging-android/

Vujovic, F. [Filip Vujovic]. (2016, mayo 29). *Firebase Cloud Messaging Push Notifications using the Android, PHP, MYSQL PART 1/2*. Recuperado de https://www.youtube.com/watch?v=LiKCEa5_Cs8

Vujovic, F. [Filip Vujovic]. (2016, mayo 30). *Firebase Cloud Messaging Push Notifications using the Android, PHP, MYSQL PART 2/2*. Recuperado de https://www.youtube.com/watch?v=MYZVhs6T_W8

10.2.8 Servicio Cámara

Chen, Bin. (2015, enero 22). Android Camera2 API Explained. Recuperado de http://pierrchen.blogspot.com/2015/01/android-camera2-api-explained.html

Google Developers Documentation. (2018). *Reference android.hardware.camera2*. Recuperado de https://developer.android.com/reference/android/hardware/camera2/package-summary

Gutknecht, Stephen. (2016, junio 29). *SimpleCamera2ServicePublish.java*. Recuperado de https://gist.github.com/RoundSparrow/142b840ca86ba7a46639f23c5c0d195b

Inducesmile. (2018). *ANDROID CAMERA2 API EXAMPLE TUTORIAL*. Recuperado de https://inducesmile.com/android/android-camera2-api-example-tutorial/

Stack Overflow. (2015, septiembre 8). *Android Camera2 front camera*. Recuperado de https://stackoverflow.com/questions/32462486/android-camera2-front-camera

Stack Overflow. (2016, diciembre 22). *Capture picture without preview using camera2 API*. Recupeardo de https://stackoverflow.com/questions/28003186/capture-picture-without-preview-using-camera2-api

Tran Nguyen, Linh. (2017, diciembre 9). *AndroidCamera2API*. Recuperado de https://github.com/eddydn/AndroidCamera2API