UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software (GIIS)

Trabajo Fin de Grado

"Un desarrollo Android para el control de dispositivos acoplados a un automóvil"

Virgilio Luis García Hoyas

Septiembre, 2015

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software (GIIIS)

Trabajo Fin de Grado

"Un desarrollo Android para el control de dispositivos acoplados a un automóvil"

Autor: Virgilio Luis García Hoyas

Directores: Pablo García Rodríguez y Andrés Caro Lindo

Tribunal Calificador:

Presidente: Manuel Barrena García

Secretario: María Luisa Durán Martín-Merás

Vocal: Pablo Bustos García de Castro

Septiembre, 2015

Dedico este proyecto a todos los que me apoyaron e hicieron posible que saliera adelante,

especialmente a mi familia y amigos, por nunca dejarme solo

y acompañarme en este camino.

AGRADECMIENTOS

Me gustaría dar las gracias en primer lugar, a mis tutores, Pablo García Rodríguez y Andrés Caro Lindo, por el tiempo y la ayuda que me han dedicado para que este proyecto saliera adelante.

También me gustaría agradecer a mi familia, a los que están y los que se han ido, el apoyo moral que me han brindado durante todos los años de mi carrera, y estar ahí siempre que se les ha necesitado.

Por otro lado, también me gustar mencionar a mis compañeros de la universidad, en particular a aquellos con los que he compartido travesía desde que entré en la carrera, por toda la ayuda y consejos que me han dado.

Por último, dar las gracias a todos mis amigos, especialmente a aquellos que, por motivos de distancia, veo unas pocas veces al año, por su apoyo incondicional y, por estar ahí siempre que se les ha hecho falta.

Resumen

El proyecto se basa en una aplicación Android, en la cual se pueden visualizar distintos datos obtenidos de forma directa de un automóvil, mediante los sensores que pueda llevar éste incorporado de fábrica, o bien, otros datos obtenidos a través de sensores que se han acoplado al coche, con el objetivo de obtener más datos de los que poder sacar un cierto provecho.

Los datos sacados directamente del automóvil, son obtenidos gracias a una aplicación implementada en *Java,* y la cual ha sido desarrollada utilizando el entorno de desarrollo *Eclipse.* Dicha aplicación se comunica con un dispositivo denominado "*ELM327 mini*", que ha sido acoplado al puerto OBD, el cual es obligatorio que esté presente en todos los coches posteriores a 1991. OBD son las siglas de "On Board Diagnostic" o lo que es lo mismo, "*diagnóstico de a bordo*", y permite monitorizar numerosos datos del vehículo.

Para el posterior uso de los datos, estos son guardados con una frecuencia concreta en una base de datos alojada en un servidor propio. De enviar estos datos a dicho servidor, se encarga una Raspberry Pi[1] que se ha incorporado al coche. Para dicho almacén de datos, se ha utilizado el sistema gestor de base de datos *PostgreSQL*.

Es la Raspberry Pi la encargada de realizar las peticiones de datos al dispositivo conectado al puerto OBD del coche, mediante una conexión Bluetooth. Para ello fue necesario incorporar un dispositivo Bluetooth a la misma.

Por otro lado, a la Raspberry Pi se le han añadido dos sensores, uno de temperatura y otro de detección de CO (monóxido de carbono). Ambos sensores recogen datos con una frecuencia concreta y son almacenados en una base de datos alojados en el servidor propio, al igual que se hace con los datos que se obtienen de forma directa del coche. Así mismo, para garantizar una cierta seguridad, si alguno de los dos sensores recoge algún valor que sobrepasa las cuotas normales, se le notificará al instante al usuario, de forma totalmente automática, mediante un mensaje a su cuenta de *Telegram*[2] por si este no está en el coche en el momento en el que este caso se diera. Todo esto se hace gracias a una serie de *scripts* implementados utilizando el lenguaje de programación *Python.* De igual modo, fue necesario el uso de la API de *Telegram*, así como el de otras herramientas relacionadas, las cuales se detallarán en apartados posteriores, para el envío de mensajes de *Telegram* mencionado anteriormente.

Todos los datos recogidos, tanto de forma directa del coche, como a través de los sensores que se han incorporado para el proyecto, pueden ser visualizados mediante la aplicación Android, desarrollada utilizando el entorno de desarrollo *Android Studio*. Dicha aplicación consultará la base de datos para acceder a los datos con los que trabaja.

El resultado final es una aplicación Android totalmente funcional, en la cual se pueden visualizar los datos del coche que se han ido almacenando. La aplicación presenta diferentes vistas dependiendo del tipo de datos que el usuario quiera consultar, aislando totalmente unos datos de otros. Asimismo, el usuario tiene un histórico de todos los datos, es decir, podrá consultar en la propia la aplicación los datos obtenidos en una fecha determinada, lo cual le

puede ser de utilidad para la ayuda de detección de ciertas averías. De igual modo se le informará al usuario de cualquier dato que se encuentre fuera de los rangos normales, quedando todo ello registrado en la base de datos.

Índice

Resumen ... 6

1. Introducción ... 13

1.1 Motivación y objetivos ... 13

1.2 Desarrollo y problemas ... 14

1.3 OBD (On Board Diagnostics) ... 15

1.3.1 Protocolos para el sistema OBD ... 15

1.4 La tecnología Bluetooth ... 16

2. Sistemas de almacenamiento ... 19

2.1 PostgreSQL ... 19

2.1.1 Límites de PostgreSQL ... 20

2.1.2 Motivos de la elección de PostgreSQL ... 20

2.2 PgAdminIII ... 21

2.2.1 Características principales ... 22

3. Herramientas Software ... 25

3.1 Eclipse ... 25

3.1.1 Características ... 26

3.1.2 Eclipse IDE para Java EE ... 26

3.2 Android Studio ... 26

3.2.1 Características ... 27

3.3 Python ... 28

3.3.2 Librerías utilizadas ... 28

3.4 Telegram Messenger ... 28

3.4.1 Ventajas ... 29

3.4.2 API ... 29

3.5 No-IP DUC ... 30

4. Hardware ... 33

4.1 Asus F550CC ... 33

4.2 LG G2 D802 ... 35

4.3 Raspberry Pi B+ ... 36

4.4 ELM327 mini ... 37

4.5 Sensor MQ-7 ... 38

4.6 Sensor DHT-11 ... 39

4.7 Adaptador Bluetooth USB ... 40

4.8 Adaptador Wi-Fi USB ... 40
5. Metodología ... 45
5.1 Estudio previo ... 45
5.2 Análisis y diseño ... 46
5.3 Implementación ... 47
5.3.1 Trabajando con los datos del coche ... 48
5.3.2 Scripts de los sensores añadidos ... 56
5.3.3 Aplicación móvil ... 59
5.4 Pruebas ... 67
6. Resultados y discusión ... 71
6.1 Datos obtenidos del coche ... 71
6.2 Datos obtenidos de sensores ... 73
7. Conclusiones y líneas futuras ... 77
7.1 Conclusiones ... 77
7.2 Líneas futuras ... 78
Anexos ... 83
A.1 Manual de usuario ... 83
A.2 Casos de uso ... 89
A.3 Glosario ... 93
Bibliografía ... 95

Capítulo 1

Introducción

1. Introducción

1.1 Motivación y objetivos

Vivimos en una sociedad cada vez más avanzada. Una sociedad en la que, cada vez más, tenemos casi todo al alcance de nuestra mano, gracias al desarrollo exponencial que están sufriendo nuevas tecnologías como los *wereables*[3]. Podríamos decir por lo tanto que vivimos en una sociedad "conectada".

Puede sonar a película futurista de ciencia ficción, pero este mismo año, en Suecia, han comenzado ya a hacer pruebas con un chip implantado debajo de la piel de una persona en cuestión, y que le permitiría abrir puertas, realizar pagos en el supermercado y otras muchas facilidades[4]. Esto mismo lo podríamos llevar a otro ámbito del día a día de la mayoría de las personas: los coches.

La industria del automóvil, desde hace unos años hasta ahora, está incorporando a sus vehículos cada vez más aplicaciones y sensores, cuyo objetivo es, principalmente, el de aumentar la seguridad de sus pasajeros. Sin ir más lejos, ya hay en el mercado un gran número de vehículos que tienen incorporados varios sensores capaces de detectar obstáculos en la carretera y aminorar la velocidad para evitar el impacto. Si esto hasta hace pocos años era casi impensable, la pregunta es ¿hasta dónde podrá llegar esta tecnología?

No podemos responder a la respuesta anterior. Sin embargo, lo que sí sabemos, es entorno a qué se va a trabajar estos próximos años, para que nuestros coches sean cada vez más seguros, y no es otra cosa que la comunicación coche a coche.

Ya en este año han comenzado las primeras pruebas reales en las cuales los coches se comunicaban entre sí, intercambiando información relativa a su posición, velocidad, trayectoria[5]... entre otras muchas cuestiones, y todo ello con el objetivo de reducir al máximo el número de accidentes. Todavía no sabemos cuándo tendremos en el mercado coches con esta tecnología incorporada, pero lo que sí sabemos es que, tarde o temprano, llegarán.

Como se ha dicho anteriormente, los coches nuevos tienen cada vez más sensores incorporados. Sin embargo, cualquiera que tenga un coche anterior al año 2000, sabrá que éstos apenas nos revelan información.

Esta fue realmente la motivación para el desarrollo del trabajo: poder combinar, de cierto modo, la tecnología existente cerca del año 2000, puesto que, particularmente, el coche del autor de este trabajo fue matriculado en el 1999, con la tecnología con la que hoy en día contamos.

Es este ámbito concreto en el que se quería trabajar, pudiendo fijar así unos objetivos claros, es decir, poder recolectar más información de un coche que apenas informa de la velocidad, revoluciones y kilómetros recorridos, e incorporar algunos sensores extra dentro del coche, que podrán dar información relevante. Y, dependiendo de esta información, poder interactuar

con el propio vehículo, de tal modo que se puedan prevenir hechos nos deseados o incluso averías futuras, teniéndolo todo ello al alcance de nuestra mano gracias a una aplicación Android.

El objetivo, por lo tanto, del proyecto es, básicamente, la posibilidad de disponer de una aplicación en nuestro móvil mediante la cual se disponga de un conjunto de datos relevantes de nuestro coche. Y, además, aprovechar de alguna manera estos datos mediante notificaciones a nuestro mismo terminal.

1.2 Desarrollo y problemas

Nada más empezar surgieron las primeras dudas. La más relevante, quizás, fue la de cuál iba a ser el elemento principal con el que se conectara el coche para extraer información, ¿tendría cabida sacar directamente la información a la aplicación móvil? o quizás, ¿sería mejor utilizar algún elemento intermediario entre el coche y la aplicación? De este modo, el elemento intermediario debería comunicarse con el automóvil y además enviar a un servidor la información obtenida de éste para que, posteriormente, la aplicación móvil pudiese acceder a estos datos. Esta duda se prolongó durante varios días. Así, para perder el menor tiempo posible, se comenzó a trabajar de forma separada sobre ambas ideas.

Finalmente, y con la ayuda de los tutores, se decidió trabajar sobre la segunda idea. Ahora había que pensar a cerca del elemento a utilizar como intermediario entre el coche y la aplicación móvil. No hubo muchas dudas y la elección fue la Raspberry Pi B+, puesto que tiene una capacidad de procesamiento que es de sobra suficiente para los requisitos del proyecto. Además, tiene un consumo mínimo, por lo que puede conectarse al coche sin problemas. Y, principalmente, porque además de utilizarla para comunicarse con el coche, se le pueden añadir sensores para recolectar más información, que era precisamente lo que desde un principio se estaba buscando.

Con todas las ideas claras se empezó a centrar el trabajo en sacar toda la información posible del coche. Pronto se descubrió que se había sido demasiado optimista a la hora de plantear objetivos. El coche con el que se trabajaba, un *Citroën Xsara* matriculado en el 99, apenas tenía sensores incorporados, y sólo se pudieron sacar 5 datos del coche, 3 de ellos irrelevantes para la idea que se había pensado. A pesar de esto, se empezó a trabajar con lo poco que se tenía, y se fue almacenando los datos de forma periódica en una base de datos que se encontraba en un servidor propio.

Una vez hecho todo lo posible con el coche, se empezó a trabajar con otros sensores, conectándolos a la Raspberry Pi. Concretamente se acopló un sensor de temperatura y otro de detección de CO. Cuando se tenían los datos necesarios, se decidió incorporar la API de *Telegram* para realizar ciertos avisos, utilizando los datos obtenidos de estos sensores.

Tras finalizar todo el trabajo que se creía oportuno con los sensores, se pudo trabajar con otro coche adquirido recientemente, un *Opel Astra* matriculado en el 2015,del cual se pudieron obtener bastante más datos con los que poder trabajar.

1.3 OBD (On Board Diagnostics)

Como sus propias siglas indican, el OBD es un sistema de diagnóstico a bordo incorporado en los automóviles y es el encargado de, por ejemplo, si falla alguna pieza del motor de nuestro coche, encender una luz de advertencia en el cuadro de mandos. En un principio, este sistema nació en el año 1988 con el objetivo de poder controlar los límites máximos de emisiones, así como un autocontrol del propio vehículo.

La primera versión fue el OBD I, surgió en Estados Unidos y se hizo obligatorio de instalar para los productores de vehículos (coches y camiones) a partir del año 1991. Sin embargo, fue un relativo fracaso, puesto que no era muy efectivo a la hora monitorizar muchos de los componentes relacionados con las emisiones. Este fue el motivo de que en el año 1996 apareciera también en Estados Unidos una segunda versión mejorada del OBD I, el OBD II, que sí cumplía finalmente con el objetivo de poder monitorizar y controlar de forma efectiva las emisiones del vehículo.

Sin embargo, en Europa no existe el OBD II como tal, sino una variación del mismo conocido como EOBD, es decir, *Diagnóstico de a Bordo Europeo*. Y que se trata de un sistema algo más sofisticado que el OBD II, en el que los componentes se adaptan por sí solos dependiendo de las condiciones del motor.

Está ya previsto que, en un futuro no muy lejano, aparezca la tercera versión de este sistema, el OBD III, y que contará con el añadido de que avisará de forma automática a las autoridades en el caso de producirse algún tipo de problema que afecte a las emisiones de gases del vehículo[6].

Hoy en día, debido a toda la evolución que el OBD ha experimentado en los últimos años, es utilizado principalmente por los mecánicos, quienes conectan al conector OBD del automóvil máquinas de diagnóstico de una gran sofisticación. Esto les permite obtener gran información acerca de las averías, ya que el sistema OBD, si se produjera algún fallo en el vehículo, guarda un registro del fallo en cuestión, así como cuáles eras las condiciones concretas que se daban en el instante del mismo.

1.3.1 Protocolos para el sistema OBD

Existen tres protocolos principales de comunicación para el OBD II y EOBD. Cada uno de ellos presenta algunas variaciones dependiendo del sistema del automóvil, así como del escáner que se utilice para obtener la información requerida.

A pesar de que en los últimos años ha habido una serie de cambios entre los fabricantes del protocolo, por lo general, éstos están distribuidos de la siguiente manera: los vehículos europeos y asiáticos utilizan el protocolo **ISO 9141**, los vehículos fabricados por General Motors usan el protocolo **SAE J1850 VPW** y, los vehículos Ford, utilizan el protocolo de comunicación **SAE J1850 PWM**.

A pesar de la existencia de estos tres protocolos, éstos son utilizados únicamente para la conexión eléctrica del OBD. Y con el objetivo de que todo sea más sencillo, el conjunto de comandos que permite comunicarse con este sistema está unificado bajo el protocolo **SAE J1979**.

En la figura 1.1 se puede ver un esquema básico del conector OBD de un automóvil cualquiera:

Figura 1.1 Terminales del dispositivo OBDII

1.4 La tecnología Bluetooth

La tecnología Bluetooth desempeña un importante papel en este proyecto, puesto que es la encargada de transmitir los datos desde el puerto OBD del coche hasta la Raspberry Pi acoplada al mismo. Pero ¿qué es la tecnología Bluetooth?, ¿cómo surgió? y ¿cuáles son las ventajas de su uso?

Bluetooth se trata de una tecnología que permite la transmisión inalámbrica de voz y datos entre diferentes dispositivos, utilizando para ello ondas de radio de corto alcance, concretamente de 2.4 GHz de frecuencia. Esta tecnología empezó a ser desarrollada por la compañía Ericsson con el objetivo de permitir comunicaciones de corto alcance. Sin embargo al poco tiempo este proyecto llamó la atención de las grandes empresas del panorama tecnológico, quienes formaron la alianza Bluetooth SIG (*Bluetooth Special Interest Group*) y siguieron desarrollando la idea para que, finalmente, en 1998 la tecnología Bluetooth viera la luz[7].

Las ventajas de esta tecnología fueron las que nos hicieron decantarnos por el uso de la misma para el proyecto. La primera ventaja es más que evidente: Bluetooth se basa en una comunicación inalámbrica, por lo que nos podemos evitar tener cables por medio del habitáculo del coche que puedan ser molestos. Otras ventajas son:

- Establece conexiones con poco gasto de energía. Esto es básico, puesto que uno de los dispositivos conectados es la Raspberry Pi.
- Se trata de una tecnología ampliamente extendida, por lo que no es complicado encontrar dispositivos con lo que poder usarla.
- Otorga una cierta seguridad de diversas maneras de cifrado de datos, además de añadir el uso de una clave de seguridad para establecer conexiones entre dispositivos.
- Tiene un bajo coste tanto de producción como de implementación.

Capítulo 2

Sistemas de almacenamiento

2. Sistemas de almacenamiento

En este apartado se verán cuáles han sido los sistemas de almacenamientos de datos utilizados para el desarrollo de este proyecto, así como las herramientas utilizadas para llevar la gestión de estos sistemas. Como motor de base de datos se ha seleccionado *PostgreSQL*, y para la administración de las bases de datos se ha utilizado su gestor, *pgAdmin-III*. A continuación veremos ambos más en detalle.

2.1 PostgreSQL

PostgreSQL, es un sistema de almacenamiento de datos relacional y de código abierto que deriva del proyecto POSTGRES. Utiliza el lenguaje SQL92/SQL99. Su principal ventaja es que agiliza la interacción de cliente, servidor y base de datos, siendo el propio *PostgreSQL* el que asume la mayor carga del trabajo en el referido a bases de datos cuando se realizan peticiones. Utiliza *multiprocesos* en vez de *multihilos,* garantizando así la estabilidad del sistema, de tal forma que, de ocurrir un fallo en alguno de los procesos, esto no afectaría de modo alguno al resto y el sistema continuará funcionando[8].

Ha sido el pionero de gran parte de los conceptos existentes hoy en día en el sistema objeto-relacional, y que se han ido incluyendo en otros sistemas de gestión de datos comerciales.

Es compatible con la mayor parte de sistemas operativos:

- Linux
- Unix (AIX, BSD, HP-UX, SGI IRIX, Os X, Solaris, Tru64)
- Windows

Se trata de una base de datos que cumple totalmente con las características ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad), tiene soporte completo para claves foráneas, vistas, uniones, disparadores, así como procedimientos almacenados. Incluye gran variedad de tipos de datos, tales como: Integer, Numeric, Boolean, Char, Varchar, Date, Interval y Timestamp. Además de esto, permite el almacenamiento de objetos binarios de gran tamaño como pueden ser imágenes, sonido o incluso vídeos.

Unos de los puntos fuertes de *PostgreSQL*, es que utiliza un sistema denominado MVCC (Acceso Concurrente Multiversión), lo cual le permite que, mientras un proceso escribe en una tabla, otros procesos distintos puedan acceder a esa misma tabla sin necesidad alguna de bloqueos. Esto lo consigue gracias a que cada usuario obtiene una versión estable del último *commit* que se realizó.

2.1.1 Límites de PostgreSQL

A continuación se verán algunos de los límites que presenta *PostgreSQL* referidos a los tamaños de almacenamiento que no pueden ser sobrepasados.

- Tamaño máximo de base de datos: El tamaño máximo es, en principio, ilimitado, dependerá del sistema del almacenamiento del equipo en el que se encuentre alojada la base de datos.
- Tamaño máximo de tabla: El límite que *PostgreSQL* soporta para este campo, es de 32 TB.
- Tamaño máximo de tupla: En este caso el tamaño máximo soportado por el sistema es de 1.6 TB.
- Tamaño máximo de campo: El tamaño máximo que *PostgreSQL* soporta de campo es de 1 GB.
- Número máximo de filas por tablas: En este caso el número de filas máximo soportados es ilimitado.
- Número máximo de columnas por tablas: Dependerá de los tipos usados, pero los valores oscilarán entre 250 a 1600.
- Número máximo de índices por tabla: El número máximo de índices soportados por tabla es ilimitado.

2.1.2 Motivos de la elección de PostgreSQL

En este apartado se verán cuáles han sido los principales motivos de la elección de *PostgreSQL* como sistema de almacenamiento de datos para el proyecto.

Algunos de los motivos fueron los siguientes:

- Se trata de un sistema de base de datos totalmente gratuito.

- A pesar que, para bases de datos no muy grandes, la velocidad no es excesivamente rápida, esta velocidad se mantiene al aumentar el tamaño de la base de datos. Otros programas en cambio disminuyen su velocidad de forma considerable al aumentar el tamaño de la base de datos.

- *PostgreSQL* ha sido diseñado para que sus costes de mantenimiento sean mucho menores que los que presentan muchos productos comerciales.

- Destaca por su estabilidad y confiabilidad.

- Es multiplataforma, como se ha mencionado anteriormente, está disponible para Linux, casi cualquier sistema Unix, así como para Windows.

- Ha sido diseñado para trabajar con altos volúmenes de datos.

- Permite copias de seguridad en caliente. Esto lo hace gracias al sistema MVCC nombrado en el apartado anterior, el cual permite a los accesos de sólo lectura continuar leyendo datos consistentes durante la actualización de registros.

- *PostgreSQL* tiene una gran capacidad de almacenamiento.

- Tiene un buen sistema de seguridad gracias a la gestión de usuarios, grupos de usuarios y contraseñas.

- Es escalable, siendo capaz de ajustarse al número de CPU, así como a la cantidad de memoria que dispone, soportando la mayor cantidad de peticiones a la base de datos que puede atender de forma correcta.

2.2 PgAdminIII

PgAdminIII es una herramienta Open Source de gran utilidad, puesto que permite crear, diseñar o administrar bases de datos *PostgreSQL* con gran facilidad, todo ello gracias a una interfaz gráfica bastante intuitiva para el usuario.

Esta herramienta cuenta con todas las funcionalidades básicas que posee un administrador de bases de datos.

La aplicación cuenta con un editor de código SQL con resaltado de sintaxis, que permite escribir consultas SQL de forma rápida. Cuenta también con editor de código procedural, así como un agente que permite lanzar *scripts*, entre otras muchas cosas. En la figura 2.1, se muestra una captura de pantalla de esta herramienta.

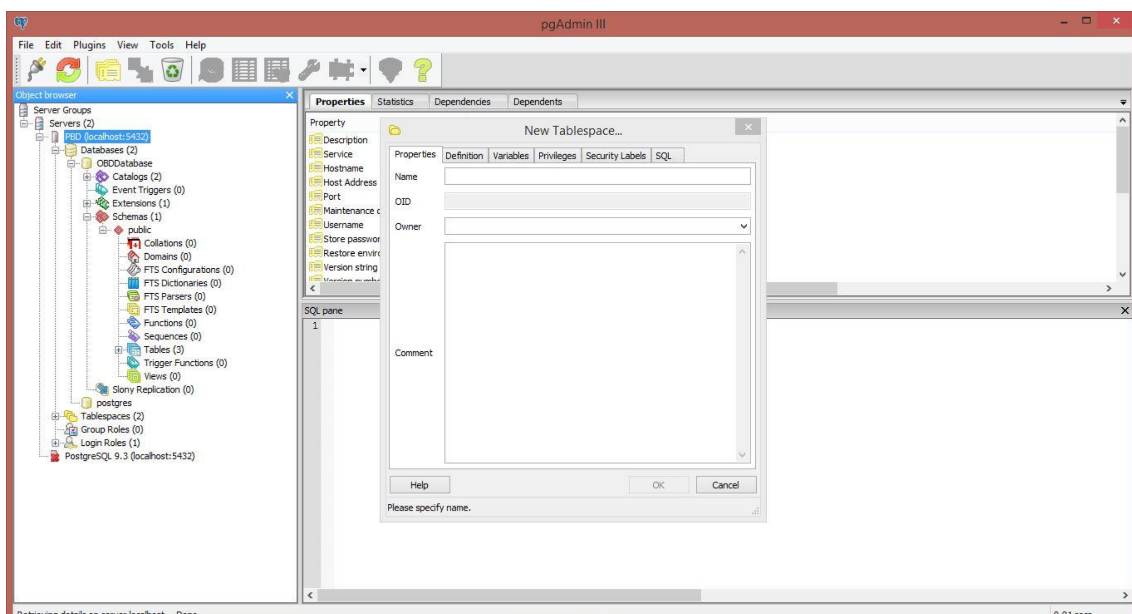


Figura 2.1 Captura de pantalla de PgAdminIII

2.2.1 Características principales

Algunas de las principales características de *PgAdminIII* y algunas de sus funcionalidades más relevantes son las siguientes:

- Posee un panel SQL el cual se pueden ver las instrucciones SQL generadas como resultado de las funcionalidades que nos ofrece la interfaz, como puede ser al crear una tabla.
- Dispone de un EXPLAIN gráfico para consultas lo cual facilita el trabajo a la hora de optimizarlas.
- Permite editar desde la propia herramienta los archivos de configuración **postgresql.conf y pg_hba.conf.**
- Facilita los procesos de restauración y *backups* gracias a un asistente gráfico que permite seleccionar el elemento que se desea restaurar, ya sean bases de datos, esquemas, tablas, etc.
- Posee una funcionalidad conocida como **Grant Wizard** que permite modificar de forma rápida y sencilla los permisos de bases de datos y tablas.

Capítulo 3

Herramientas software

3. Herramientas Software

En este apartado se van a explicar de forma breve, las herramientas software utilizadas durante el desarrollo del proyecto. La primera que veremos, será Eclipse, concretamente la última versión, el cual se ha utilizado para el desarrollo de la aplicación encargada de conectarse con el dispositivo ELM327 mini acoplado al puerto OBD del coche para poder obtener los datos deseados. La siguiente herramienta que se va a explicar es *Android Studio*, en su versión 1.2.2, utilizada en este caso para llevar a cabo la generación Android. Por otra parte se explicará que es *Python,* cuya versión empleada ha sido la 2.7, el cual se utilizó para manejar los sensores acoplados a la Raspberry Pi, así como para trabajar con la API de *Telegram.* También se verá dicho servicio de mensajería, *Telegram,* junto a su API, utilizada para enviar mensajes de aviso al terminal móvil del usuario, cuando se dan unas condiciones concretas. Por último se explicará la herramienta *No-IP DUC*, utilizada para asociar a la IP del servidor donde se encuentra la base de datos, a un nombre de dominio.

Cabe mencionar también, que el ordenador con el cual se ha trabajado con todas estas herramientas, tiene instalado el sistema operativo *Windows*, aun así, al tratarse de herramientas multiplataforma, no habría problema alguno en migrar a otro sistema operativo.

3.1 Eclipse

Eclipse es un entorno de desarrollo software multi-lenguaje, de código abierto y multiplataforma, desarrollado casi en su totalidad en *Java*, concretamente un 92,66% de las líneas de código de su última versión están escritas en dicho lenguaje. Básicamente se trata de un entorno de desarrollo integrado (IDE) que reúne un amplio número de herramientas y funciones básicas para el desarrollo de aplicaciones software. No ha sido pensado para desarrollar en un lenguaje específico, aunque es popular dentro de la comunidad de desarrolladores Java, puesto que la versión estándar de Eclipse viene con el *plug-in*[10] JDT (Java Development Toolkit) incluido. Además de esto, dispone de varios *plug-ins* que permitirán al desarrollador programar en otros lenguajes tales como Ada, C/C++, Erlang, COBOL, CORBA, Haskell, Groovy, Javascript, Perl, Prolog, PHP, Python, Ruby y Scala entre otros[9].

Actualmente, Eclipse cuenta con hasta 13 versiones disponibles, para poderse adaptar lo máximo posible a las necesidades del usuario. Además cada versión puede ser ampliada gracias a un amplio listado de *plug-ins* disponibles para el usuario.

Cabe destacar también que si alguna de las versiones no se adaptan a las necesidades del usuario todo lo que este quisiera, existe una amplia lista de herramientas software basadas en el propio Eclipse, que pueden satisfacer unas necesidades mucho más concretas.

3.1.1 Características

Las principales características de Eclipse pueden ser resumidas y enumeradas de la siguiente manera:

- Dispone de un editor de texto con analizador sintáctico.

- La compilación es en tiempo real.

- Posee pruebas unitarias con JUnit, control de versiones con CSV, integración con Ant y asistentes para llevar a cabo creaciones de proyectos, clases, tests, y refactorización

- Del mismo modo, gracias a los plug-ins que tiene la herramienta a disposición del usuario, es posible añadir un control de versiones mediante *Subversion* e integración con *Hibernate*.

3.1.2 Eclipse IDE para Java EE

El paquete concreto de Eclipse con el que se ha trabajado ha sido *Eclipse IDE Java EE*, el cual contiene un conjunto de herramientas específicas para el desarrollo de aplicaciones en java. Algunas de las características que incluye esta versión son:

- Contiene herramientas para el desarrollo Java y Java EE.

- Incluye herramientas para el desarrollo en Javascript.

- Integración Maven para Eclipse

- Incluye también la herramienta Git

- Explorador de sistemas remotos

- Editor XML

3.2 Android Studio

Android Studio es un entorno de desarrollo integrado (IDE) para Android, lanzado por Google, que remplazó a Eclipse como el IDE de desarrollo oficial para aplicaciones Android. Está basado en *IntelliJ IDEA*[11], un IDE para Java de *Jetbrains* sobre el cual han desarrollado una serie de características específicas para el desarrollo de aplicaciones Android[12]. En la figura 3.1, podemos ver una captura de pantalla del *Android Studio*.

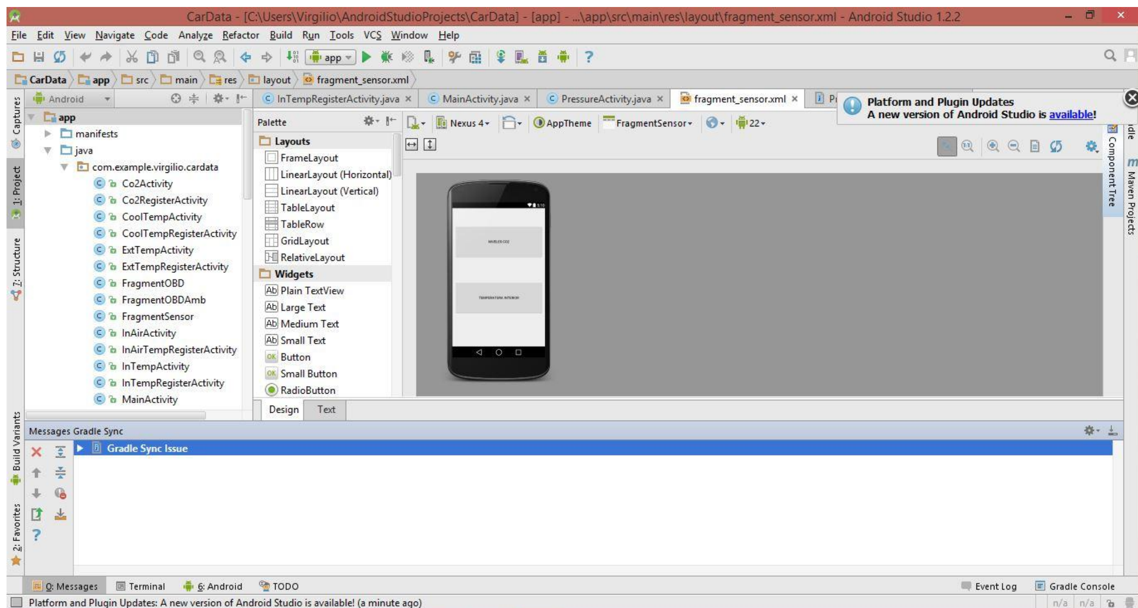


Figura 3.1 Captura de pantalla de Android Studio

Se trata de un software multiplataforma y además es publicado de forma gratuita bajo una Licencia apache 2.0.

3.2.1 Características

Android Studio posee una serie de características bastantes ventajosas con respecto a Eclipse como IDE de desarrollo de aplicaciones Android, las cuales han motivado que este software sustituya a Eclipse como aplicación oficial en este campo. Las principales características de *Android Studio* son las siguientes:

- Renderización en tiempo real.

- Posee una consola de desarrollador, con consejos de optimización, ayuda para la traducción, y estadísticas de uso.

- Incluye soporte para construcción basada en Gradle[13], lo cual presenta bastantes ventajas, como pueden ser la facilidad a la hora de la reutilización de código y recursos, la facilidad de configurar, extender y personalizar el proceso, entre otras muchas ventajas.

- Usa herramientas *Lint* (herramientas encargadas de realizar un análisis de del código fuente) para detectar problemas de rendimiento, usabilidad y compatibilidad de versiones entre otros.

- Plantillas para crear diseños comunes de Android.

- Incluye también soporte para programar aplicaciones para Android Wear[14].

3.3 Python

Básicamente *Python,* no es otra cosa más que un lenguaje de programación interpretado, cuya filosofía se basa en que su sintaxis favorezca a que el código sea fácilmente legible. Se trata de un lenguaje multiparadigma[15], que usa tipado dinámico[16] y es multiplataforma. Así mismo, posee una licencia de código abierto[17].

Una de las características más importante de *Python,* es su facilidad de extensión, es decir, se pueden escribir nuevos módulos fácilmente en C o C++.

3.3.2 Librerías utilizadas

A continuación se verán de forma breve aquellas librerías para *Python,* utilizadas durante el desarrollo del proyecto:

- **sys:** provee acceso a funciones y objetos mantenidos por el intérprete.
- **time:** permite obtener la hora y fecha actual.
- **Adafruit_DHT:** facilita el manejo del sensor DHT_11.
- **os:** permite acceder a funcionalidades para interactuar con el sistema operativo.
- **psycopg2:** se trata de un adaptador para *PostgreSQL.*
- **pexpect:** permite el control de otras aplicaciones.

Es importante mencionar en este apartado, que durante la realización del proyecto no fue necesaria la utilización de ningún entorno de desarrollo para *Python.*

3.4 Telegram Messenger

Telegram es un servicio de mensajería por internet basado en la nube. Inicialmente fue desarrollado únicamente para usarse en teléfonos móviles. Sin embargo, actualmente se trata de una herramienta multiplataforma. Es, además, una aplicación totalmente gratuita cuyos puntos fuertes son su velocidad y seguridad.

Su diseño de arquitectura en la nube dota de una gran importancia a la privacidad y seguridad. Para garantizar estos dos aspectos, se llevó a cabo el desarrollo de un protocolo exclusivo para el procesamiento de datos.

Se trata, además, de una aplicación libre, aunque por ahora sólo en el lado de la aplicación. La parte del servidor seguirá cerrada hasta que sus desarrolladores puedan crear una arquitectura descentralizada dentro de su propio servicio de la nube.

Para el desarrollo de otras aplicaciones, *Telegram* pone a disposición de los usuarios una API, para evitar así que no se cumpla la guía de seguridad establecida por su equipo de desarrollo[18].

3.4.1 Ventajas

La elección de *Telegram* como aplicación encargada del envío de mensajes al usuario para este proyecto no fue casual. Las características principales que hicieron elegir *Telegram* sobre otros servicios de mensajería más conocidos fueron las siguientes:

- Los mensajes está fuertemente cifrados.
- Gracias a su sistema de la nube, permite acceder a los mensajes desde varios dispositivos distintos.
- Destaca por su rapidez a la hora de la entrega de mensajes.
- Posee un gran número de servidores repartidos por todo el mundo lo cual hace dota a la aplicación de una mayor seguridad y velocidad.
- Es gratuita, de código abierto y ofrece una API de gran utilidad.
- No tiene límites a la hora del envío de mensajes.

3.4.2 API

Como ya se ha mencionado anteriormente, *Telegram* ofrece una API para diferentes plataformas, que permite a los usuarios poder crear sus propios clientes de *Telegram* de forma totalmente personalizada. Esto nos va a permitir asociar y acceder al servicio que tenemos en nuestro teléfono móvil, desde cualquier otra plataforma. En concreto, en este proyecto, el cliente es instalado en una Raspberry Pi B+ y posteriormente se asocia con la cuenta del teléfono móvil del usuario. De este modo ya se podrá interactuar con el servicio *Telegram* del usuario desde la propia Raspberry Pi.

3.5 No-IP DUC

No-IP DUC se trata de una herramienta perteneciente al servicio No-IP. No-IP es un servidor que permite crear dominios para redireccionar conexiones. De tal modo que convierte nuestra dirección IP en un nombre de dominio elegido por el usuario. Esto es de gran utilidad, puesto que además de por la comodidad evidente, de hacer referencia al servidor a través de un nombre de dominio y no de una IP, el nombre de dominio queda asociado al servidor a pesar de que este tenga una IP dinámica y que la misma cambie cada cierto tiempo.

No-IP DUC lo que hace es automatizar el proceso descrito anteriormente, y se encarga de redireccionar de forma automática la conexión a nuestro servidor. En la figura 3.2, se muestra una captura de pantalla de la herramienta.

Figura 3.2 Captura de pantalla de No-IP DUC v4.1.0

Capítulo 4

Hardware

4. Hardware

En este apartado se explicarán los detalles técnicos de cada uno de los dispositivos hardware utilizados para el desarrollo de su proyecto.

4.1 Asus F550CC

Éste ha sido el ordenador portátil utilizado para el desarrollo del proyecto. Es un ordenador de gama media. Las herramientas software necesarias para la realización del proyecto instaladas en este dispositivo son:

- Eclipse Java for EE
- Android Studio
- PostgreSQL
- PgAdminIII
- No-IP DUC

Las características técnicas de este ordenador portátil son las siguientes[19]:

Procesador	Modelo: Intel Core i7-3537U Velocidad de reloj: 2.00 GHz Caché: 4 MB
Memoria RAM	Estándar: 4 GB Ampliable: 8 GB Tipo: DDR3 1600MHz
Almacenamiento	Capacidad de Disco: 500GB SATA Nº de discos duros: 1 Velocidad de rotación: 5400 rpm
Tarjeta Gráfica	Capacidad: 2GB Modelo: NVIDIA GeForce GT 720M DDR3 Dedicada 2048x1536
Pantalla	Calidad de imagen: 1366x768 Formato: 16:9 Tamaño: 15.6" Tipo: Retroiluminación LED Slim Glare HD
Unidad óptica	Tipo de lector: DVD 8X Supermulti, Doble Capa [SATA] (Integrada)
Audio	Altavoces: Estéreo integrados
Conexiones	Audio: 1 x Entrada/Salida línea audio (combo) 1 x Micrófono integrado Lector de tarjetas: SD (SDHC/SDXC) Puertos: 1 x USB 2.0 1 x USB 3.0

	Vídeo: 1 x HDMI 1 x VGA (D-Sub)
Conectividad	Ethernet: 10/100/1000 Mbps WiFi: 802.11 bgn
Batería	Duración máxima: 3h 30min Nº de celdas: 4
Software	Sistema operativo: Windows 8.1 Original 64 bits Software preinstalado: Adobe Reader ASUS AIRecovery ASUS ATK Package ASUS Color Enhancement ASUS DVD ASUS Installation Wizard ASUS KBFilter driver ASUS Live Frame III ASUS Live Update ASUS Product Registration Program ASUS SceneSwitch ASUS Win Flash IceCool Technology Instant on McAfee Internet Security NVIDIA Optimus technology Power4Gear Hybrid Skype SonicMaster Technology Super Hybrid Engine II Virtual Camera
Teclado y ratón	Teclado integrado: Teclado tipo chicle con teclado numérico Touch pad: Touchpad multitáctil [16:9]
Características físicas	Dimensiones: 38 x 25,1 x 2,48 ~ 3,17 cm Peso: 2.3 Kg
Coste	599 €

Figura 4.1 Imagen del ordenador portátil

4.2 LG G2 D802

Este fue el terminal móvil utilizado durante el desarrollo de la aplicación Android, cuyo objetivo principal no era otro que el de poder realizar pruebas de la aplicación. Estas pruebas se desarrollaron tanto durante la fase de desarrollo, pudiendo así ir depurando el código con mayor facilidad, como una vez finalizada la aplicación, probando el correcto funcionamiento de ésta. Durante la fase de desarrollo no es imprescindible la utilización de este dispositivo móvil, puesto que la herramienta *Android Studio* posee su propio emulador para poder ir depurando el código. A pesar de esto, la enorme lentitud del mismo dificulta bastante el poder realizar las pruebas de una forma eficaz, y por eso mismo se optó por usar este dispositivo móvil. Por último, el propio móvil era el encargado de dotar de conexión a Internet a la Raspberry Pi, gracias a su función de punto de acceso Wi-Fi, para que esta pudiera enviar los datos obtenidos al servidor.

Algunas de las características técnicas de este dispositivo son las siguientes[20]:

Especificaciones básicas	Sistema operativo: Android 4.4.2, KitKat Velocidad del procesador: 2.26GHz Quad-Core Qualcomm Snapdragon Batería: SiO 3000mAh Banda RF: LTE Cat.4 800/900/1800/2600 - HSPA+ 42Mbps 900/1900/2100 Tiempo en reposo, Máx(hrs): Hasta 826Hs (DRX7)
Dimensiones	138.5 x 70.9 x 9.14 mm
Pantalla	Tamaño: 5.2" Tipo: Full HD IPS (1800 x 1920 pixels / 423 ppi) Táctil: Full Touch
Conectividad	WiFi: Sí (802.11 bgn) WiFi Directo: Sí DLNA : Sí (+ Wi-Fi Direct) NFC: Sí A-Gps: Sí GPS: Sí
Audio/Vídeo	Codec de vídeo: MP3, AAC, AAC+, H.263 profile0, H.263 profile3, H.263 AVC, H.264, MPEG-4, MPEG-4a, VP8, WMV7, WMV8, WMV9S, WMV9M Grabación: Sí, 720p @60fps Radio FM: Sí MP3: Sí Grabación Dual: Sí
Cámara	Cámara principal: 13 MP con AF Flash: LED Cámara frontal: 2.1MP (W2M 1920x1080)
Mensajería	SMS en cadena: Sí E Mail: Sí
Características	Memoria Interna: 16GB

avanzadas	Memoria RAM: 2GB LPDDR3 800MHz A-GPS/GPS: Sí Función Quickmemo: Sí Función QRemote: Sí Bluetooth: Sí Acelerómetro: Sí Radio FM: Sí Mp3: Sí
Internet	Wi-Fi: Sí (802.11 bgn)
Coste	327 €

Figura 4.2 Imagen del terminal Android

4.3 Raspberry Pi B+

La Raspberry Pi utilizada para la realización de este proyecto ha llevado a cabo varias funciones. La primera de ellas es la de obtener la información deseada del coche, a partir del dispositivo conectado a este (ELM327 mini), para enviarla posteriormente a la base de datos alojada en nuestro servidor. Para ello fue necesario acoplarle un dispositivo Bluetooth con el objetivo de que así pudiera conectarse al ELM327 mini acoplado al coche, así como un adaptador USB Wi-Fi para poder dotar a la propia Raspberry Pi de conexión a Internet mediante el teléfono móvil para que esta pudiera enviar los datos al servidor.

Además de esto, se incorporaron a la propia Raspberry Pi, un sensor de temperatura y otro de detección de CO. La función de la Raspberry Pi es, en este caso concreto, la de enviar los datos recogidos a partir de estos sensores, e informar al usuario mediante un mensaje a su cuenta de *Telegram*, si alguno de los valores sobrepasan unos ciertos límites.

Los detalles técnicos de la Raspberry Pi utilizada durante el desarrollo de este proyecto pueden observarse en la siguiente tabla[21]:

Modelo	Raspberry Pi B+
Tipo de	CPU ARM1176JZF-S 700 MHz

procesador	
Memoria RAM	512 MB
Conexiones	4 x USB 2.0 1 x Salida audio mini jack 3.5 mm 1 x Salida audio/vídeo HDMI 1 x Micro USB 1 x RJ45 10/100 Ethernet RJ45 Slot MicroSD para tarjetas
Conectividad	LAN Red local 10/100 WiFi (mediante adaptador USB WiFi compatible, no incluído)
Alimentación	5V/600 mA (3.5 W) via microUSB
Dimensiones	85.6 mm x 53.98 mm
Coste	30 €

Figura 4.3 Imagen de la Raspberry Pi B+

4.4 ELM327 mini

El ELM327 mini ha sido el dispositivo utilizado para obtener los datos deseados directamente del coche, tales como temperatura exterior, revoluciones actuales del propio automóvil, velocidad actual del vehículo, entre otros muchos tipos de datos. Sin embargo, la cantidad de datos que se pueden obtener, varía de forma muy significativa dependiendo del coche al que se conecte y de la antigüedad del mismo. Se trata de un artilugio dotado de conexión Bluetooth y que se conecta al coche a partir del puerto OBDII de éste. Se puede decir, por lo tanto, que es un dispositivo de autodiagnóstico, que ha sido creado con el fin de conectarse a un ordenador portátil o a un teléfono Android para poder visualizar en éstos los datos deseados del coche al que está conectado. Sin embargo, en este caso en concreto se comunica con la Rasbperry Pi, gracias a una aplicación Java que se ejecuta en la misma y que ha sido desarrollada para este proyecto.

Algunas de las especificaciones del dispositivo ELM327 mini son las siguientes[22]:

Tipo	Lector de códigos y herramienta de escaneo
Protocolo de salida	Bluetooth
Voltaje	12V
Corriente	45mA
Frecuencia onda	38400Hz
Alcance aplicable	5 -10m
Tamaño	4.7 * 2.3 * 3.1cm
Peso	34g
Coste	7€

Figura 4.4 Imagen del conector OBDII

4.5 Sensor MQ-7

El MQ-7 es un sensor de monóxido de carbono, de una alta sensibilidad y respuesta rápida, propicio para detectar las concentraciones de este gas en el aire. La tarea que desempeña en este proyecto es la de controlar en todo momento las concentraciones de CO en el interior del coche, para poder avisar al usuario en caso de que estas sean elevadas.

Las características técnicas principales de este sensor se pueden ver en la siguiente tabla[23]:

Alimentación	5V
Tipo de interfaz	Analógico
Rango de detección	20 a 2000 ppm (partes por millón)
Temperatura de funcionamiento	-10 a 50ºC
Consumo de potencia	<750 mW
Coste	3€

Figura 4.5 Imagen del sensor MQ-7

4.6 Sensor DHT-11

El sensor DHT-11 es un dispositivo que permite medir con una gran fiabilidad tanto la temperatura como la humedad del ambiente. Aunque para este proyecto únicamente nos fijamos en la temperatura, puesto que lo que queremos es tener controlada la misma en el interior del coche para poder avisar al usuario en el caso de que sobrepasara unos ciertos valores.

A continuación se muestra de forma breve una tabla con las especificaciones técnicas de este dispositivo[24]:

Alimentación	3Vdc ≤ Vcc ≤ 5Vdc
Señal de salida	Digital
Rango de medida de Temperatura	De 0 a 50ºC
Precisión de Temperatura	±2 °C
Resolución de Temperatura	0.1ºC
Rango de medida de humedad	De 20% a 90% RH
Precisión de humedad	4% RH
Resolución de Humedad	1% RH
Tiempo de sensado	1s
Tamaño	12 x 15.5 x 5.5mm
Coste	1€

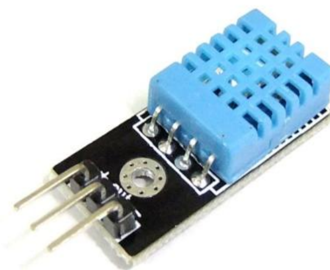


Figura 4.6 Imagen del sensor DHT-11

4.7 Adaptador Bluetooth USB

Debido a que el dispositivo ELM327 mini encargado de obtener los datos del coche funciona mediante una tecnología Bluetooth, ha sido necesario acoplar a la Raspberry Pi un adaptador Bluetooth USB con el objetivo de que ésta pueda comunicarse con el ELM327.

Características:

Versión	USB 2.0
Transferencia de datos	3MPB/s
Rango	0-100 m
Coste	5€

Figura 4.7 Imagen del adaptador Bluetooth USB

4.8 Adaptador Wi-Fi USB

La Raspberry Pi B+ únicamente puede conectarse a Internet por cable. Si se desea una conexión vía Wi-Fi es necesario acoplar a la misma un adaptador Wi-Fi USB. Como necesitamos que la Rasbperry tenga en todo momento conexión a Internet para poder, entre otras cosas, enviar de forma continua los datos que va recogiendo del coche, al servidor, se le ha acoplado un nano adaptador Wi-Fi USB.

Las características del mismo son las siguientes:

Versión	USB 2.0
Velocidad	Hasta 150 Mbps
Compatibilidad	Compatible con 802.11 bgn

Frecuencia	2.4 GHz
Dimensiones	18.6 x 15 x 7.1 mm.
Peso	2.1 gr
Coste	5€

Figura 4.8. Imagen del adaptador Wi-Fi USB

Capítulo 5

Metodología

5. Metodología

En este nuevo apartado veremos la metodología que se ha seguido para llevar a cabo todo el desarrollo del proyecto. Básicamente las tareas siguieron un ciclo de vida estructurado. Primeramente se realizó un estudio previo, donde se marcaron los objetivos que, principalmente, consistían en definir la idea que se quería llevar a cabo y, tras esto, identificar los posibles caminos que pudieran conducir al correcto cumplimiento de la idea. Tras esta fase inicial, se llevó a cabo la fase de análisis, en donde se evaluaron los requisitos que se habían marcados para, una vez evaluados, comenzar a desarrollar todo el entramado que constituye el proyecto. Posteriormente, se realizó el diseño de todo el conjunto que forma el proyecto, para así transformar estos requisitos, en una especificación bien estructurada. Una vez realizadas todas las tareas anteriores, se llevó a cabo la implementación y, tras esta, únicamente quedaba realizar las pruebas y poner a funcionar todo el proyecto.

5.1 Estudio previo

En el inicio, lo primero que se hizo fue plantear el problema por el cual se ideó el proyecto. Éste era el de intentar sacar los máximos datos posibles de coches que, debido principalmente a su antigüedad, apenas informaban de nada en su cuadro de mandos. Y, además, poder tenerlo todo en la palma de la mano, gracias al desarrollo de una aplicación para móviles Android. Así mismo, se le incorporarían, de algún modo, sensores externos al propio coche, y estos mismos datos se irían almacenando para su estudio futuro. Todo esto con el objetivo principal de poder visualizar aspectos relevantes del estado de cualquier coche antiguo, tal y como sucede en la actualidad con los nuevos automóviles

Esta idea inicial se vio algo alterada al poco tiempo, puesto que tras realizar este estudió e investigar bien acerca del problema, surgió el inconveniente de que la mayoría de los automóviles antiguos, apenas tenían sensores incluidos por lo que no se podía sacar datos realmente relevantes.

Finalmente, y tras comprobar que en los coches actuales se le podía sacar mucho más partido a esta idea, se alteró de algún modo el objetivo que se había marcado desde un principio, y este pasó a ser, el de obtener los máximos datos posibles del coche y de esta manera poder tener un historial de aquellos que fueran relevantes, para poder consultarlos en caso de averías y poder dar con la solución más fácilmente, o incluso para poder prevenir averías futuras. Además de esto se incorporó la idea de avisar mediante *Telegram* cuando ciertos valores del interior del coche fueran anómalos.

Desde un principio, la idea clara era la de trabajar con el dispositivo ELM327 mini, conocido con anterioridad, y conectarlo con una Raspberry Pi para poder sacar el máximo partido a los datos. Así mismo y aprovechando que la Raspberry Pi ya formaba parte del proyecto, fue clara la idea de incorporar a esta, los sensores que se pretendían acoplar al propio coche.

5.2 Análisis y diseño

Para la fase de análisis, lo primero que se hizo, fue abordar los requisitos que se habían planteado con anterioridad, para intentar así optimizar lo máximo posible la fase de desarrollo. Una vez decididos los datos que se querían obtener del coche, así como la manera de recogerlos y cómo almacenarlos, se abordaron los requisitos iniciales que debía cumplir aplicación Android. A grandes rasgos fueron los siguientes:

- Diferentes pantallas según el tipo de dato.
- Poder acceder siempre al valor actual de un dato determinado de forma sencilla.
- Poder acceder al historial de todos los datos.
- Poder consultar los valores de datos determinados según una fecha concreta.

Asimismo, también se fijaron los requisitos relacionados con el desarrollo de la aplicación encargada de los sensores externos al coche:

- Recoger datos de forma continua con una frecuencia determinada.
- Avisar al usuario cuando el valor de un dato fuese anómalo.

Tras esto se abordó todo lo relacionado con la base de datos, así como la elección de cuál sería su ubicación. Fue necesario, en esta fase, definir toda la estructura de esta base de datos, en la que irían almacenados todos los valores recogidos, tanto directamente del coche, como a partir de los sensores acoplados al mismo mediante la Raspberry Pi. Esta estructura fue diseñada en el sistema de almacenamiento *PostgreSQL*.

Del mismo modo y tras definir todo lo relacionado con la base de datos, se empezó a dar forma a los módulos encargados de la inserción y obtención de datos para cada tabla, estableciendo cuales serían los parámetros de los mismos.

Para la etapa de diseño se realizó un prototipo aproximado de lo que serían las diferentes pantallas que tendría la aplicación Android. En la figura 5.1, se muestra una captura de pantalla real de la aplicación, perteneciente a unos de los *Fragments*[25] principales.

Figura 5.1 Captura de la pantalla de motor

El código asociado a este *Fragment* es bastante simple como se puede ver en las siguientes líneas:

```
<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="com.example.virgilio.cardata.FragmentOBD">


<Button
        android:layout_width="match_parent"
        android:layout_height="120dp"
        android:text="Temperatura Aire Entrada"
        android:id="@+id/button_AirTemp"
        android:layout_gravity="center_horizontal|top" />

<Button
        android:layout_width="match_parent"
        android:layout_height="120dp"
        android:text="Temperatura Refrigerante"
        android:id="@+id/button_CoolTemp"
        android:layout_gravity="center" />

<Button
        android:layout_width="match_parent"
        android:layout_height="120dp"
        android:text="Tiempo Motor Encendido"
        android:id="@+id/button_TimeStart"
        android:layout_gravity="left|bottom" />
</FrameLayout>
```

Como se puede observar únicamente están definidos los botones. Esto es debido a que la *Toolbar*[26] está definida en el propio *MainActivity*[27], y la barra de los *Fragments* la maneja un adaptador que tiene su propia clase.

5.3 Implementación

En este apartado, se verá todo lo relacionado con la fase de implementación correspondiente a la aplicación principal, así como de otros métodos necesarios para el correcto funcionamiento de la misma.

Podemos distinguir, por lo tanto, varias fases bien diferenciadas. La primera fase consistió en la implementación de la aplicación *Java* que se ejecuta en la Raspberry Pi, y que es la encargada de obtener los datos deseados del coche, a través del ELM327 mini acoplado al mismo. En esta misma fase, también se implementaron los métodos para insertar los datos recogidos en la base de datos.

Tras esto, se acoplaron los sensores de temperatura y detección de CO a la Raspberry Pi, y se realizaron los *scripts* pertinentes, para obtener los datos y enviarlos a la base de datos. Así, en

esta fase fue necesario, de nuevo, implementar los métodos encargados de insertar estos datos en la base de datos. En esta mismo periodo, también se implementaron lo métodos encargados de enviar avisos a la cuenta *Telegram* del usuario, en caso de que los valores recogidos de alguno de los dos sensores, fueran anómalos.

Por último, como ya podíamos tener y por lo tanto acceder a los datos recogidos tanto del coche, como de los sensores acoplados a este, se comenzó la fase de desarrollo de la aplicación Android con la cual se podría visualizar todos estos datos. En esta fase, además del desarrollo de la propia aplicación, fue necesario implementar los métodos encargados de obtener los datos almacenados en la base de datos del proyecto.

A continuación, se explicará de forma más detallada cada una de estas fases nombradas anteriormente.

5.3.1 Trabajando con los datos del coche

Como se ha mencionado de forma breve en el apartado anterior, en esta fase se llevó a cabo el desarrollo de la aplicación *Java* encargada de obtener los datos del coche, y enviarlos al servidor. Esta fase consta de dos partes, que corresponden a las dos clases implementadas durante el desarrollo de esta parte del proyecto.

5.3.1.1 Obteniendo los datos

La primera clase se encarga de dos funciones básicas: la inicial es la de conectarse al dispositivo Bluetooth encargado de obtener la información del coche. Para ello básicamente, lo que hace es buscar a partir del dispositivo *Bluetooth* local, que sería el que se ha acoplado a la Raspberry Pi, todos los dispositivos *Bluetooth* que se encuentran a su alcance, tras esto, comprueba si la dirección física de alguno de ellos coincide con la del ELM327 (la cual se había obtenido en pruebas anteriores), en caso afirmativo, busca los servicios que ofrece el dispositivo y conecta con este. A continuación se puede ver el fragmento de código encargado de todo lo anterior:

```
OBDCom BT = newOBDCom();

//Obtiene la información del dispositivo BT local
LocalDevice BTLocal = LocalDevice.getLocalDevice();
DiscoveryAgent BTExt = BTLocal.getDiscoveryAgent();

//Realiza la búsqueda de dispositivos
BTExt.startInquiry(DiscoveryAgent.GIAC, BT);

//Espera necesaria para realizar la búsqueda dispositivos
try {
	synchronized (espera) {
	espera.wait();
```

```
        }
} catch (InterruptedException e) {
        e.printStackTrace();
}

if (ListaBT.size() == 0)

        System.out.println("No se encontraron dispositivos");



else {


        UUID[] uuid = new UUID[1];
        uuid[0] = new UUID("1101", true);


        boolean enc=false;
        String addresBT="000D183A6789";
        RemoteDevice aux= (RemoteDevice) ListaBT.elementAt(0);
        int b=1;
        while(!enc){
                aux = (RemoteDevice) ListaBT.elementAt(b-1);
                System.out.println(aux.getBluetoothAddress());

                if(aux.getBluetoothAddress().contentEquals(addresBT)){

                        enc=true;
                                }
                else {
                        b++;
                }
        }


        //Buscamos los servicios
        BTExt.searchServices(null, uuid, aux, BT);

        //Espera necesaria para realizar la busqueda de servicios
        try {

                synchronized (espera) {
                espera.wait();
        }
        } catch (InterruptedException e) {
                e.printStackTrace();
        }

        if (URL == null)
                System.out.println("ERROR URL, EL DISPOSITIVO NO SOPORTA SPP");
        else {


                //Establecemos la conexion a partir de la URL SPP
                StreamConnection conexion = (StreamConnection)
        Connector.open(URL);

                ...
        }
```

Una vez conectado con el dispositivo, lo único que queda por hacer, es abrir un flujo de escritura, y otro de lectura, entre el dispositivo local y el dispositivo acoplado al coche. Es entonces, cuando mediante el flujo de escritura, se envían los comandos deseados, dependiendo del dato que queramos obtener, y mediante el flujo de lectura almacenamos la respuesta en la base de datos. Antes de ver el código debemos entender bien cómo funciona

la comunicación con el ELM327 mini. Básicamente este dispositivo responde a una serie de códigos conocidos como OBD-II PIDS.

Estos códigos están compuestos por 4 dígitos, los dos primeros indican el modo en que vamos a trabajar, en nuestro caso siempre serán 01, puesto que hace referencia a los datos actuales. Los dos últimos números variarán dependiendo de cuál sea el dato que se quiere solicitar. A continuación se puede ver una parte del listado obtenido de la propia *Wikipedia,* en el que se indica qué dato corresponde a cada PID, así como otros datos relevantes de los mismos, como puede ser su fórmula, es decir, como hay que operar con el dato una vez obtenido.

PID (hex)	Data bytes returned	Description	Min value	Max value	Units	Formula[a]
00	4	PIDs supported [01 - 20]				Bit encoded [A7..D0] == [PID $01..PID $20] See below
02	2	Freeze DTC				
03	2	Fuel system status				Bit encoded. See below
04	1	Calculated engine load value	0	100	%	A*100/255
05	1	Engine coolant temperature	-40	215	°C	A-40
0A	1	Fuel pressure	0	765	kPa (gauge)	A*3
0B	1	Intake manifold absolute pressure	0	255	kPa (absolute)	A
0C	2	Engine RPM	0	16,383.75	rpm	((A*256)+B)/4

0D	1	Vehicle speed	0	255	km/h	A
0E	1	Timing advance	-64	63.5	° relative to #1 cylinder	(A-128)/2
0F	1	Intake air temperature	-40	215	°C	A-40
10	2	MAF air flow rate	0	655.35	grams/sec	((A*256)+B) / 100
11	1	Throttle position	0	100	%	A*100/255
1F	2	Run time since engine start	0	65,535	seconds	(A*256)+B
20	4	PIDs supported [21 - 40]				Bit encoded [A7..D0] == [PID $21..PID $40] See below
21	2	Distance traveled with malfunction indicator lamp (MIL) on	0	65,535	km	(A*256)+B
40	4	PIDs supported [41 - 60]				Bit encoded [A7..D0] == [PID $41..PID $60] See below
41	4	Monitor status this drive cycle				Bit encoded. See below
42	2	Control module	0	65.535	V	((A*256)+B)/1000

		voltage				
43	2	Absolute load value	0	25,700	%	((A*256)+B)*100/255
44	2	Fuel/Air commanded equivalence ratio	0	2	N/A	((A*256)+B)/32768
45	1	Relative throttle position	0	100	%	A*100/255
46	1	Ambient air temperature	-40	215	°C	A-40

El listado anterior, es sólo una breve muestra del listado real, ya que se han omitido gran cantidad de PIDS puesto que el listado era muy amplio, únicamente se muestra para hacernos una idea de cómo se trabaja con estos comandos.

Una vez visto esto, ya se tiene cierta noción acerca del tema, y se podrá entender con mayor facilidad el código encargado de obtener los datos del coche.

```
else {
	...
	String atm, temp, intemp, cooltemp, starttime;
	int atm_i, temp_i, intemp_i, cooltemp_i, starttime_i;

	//Abrimos y guardamos el flujo de salida en la variable
	OutputStream conexEscritura = conexion.openOutputStream();

	//Indicamos que escriba sobre el flujo de salida */
	PrintWriter enviarComando = newPrintWriter(
			new OutputStreamWriter(conexEscritura));

	//Abrimos y guardamos el flujo de entrada en la variable
	InputStream conexionLectura = conexion.openInputStream();

	//Creamos un buffer de lectura para almacenar la respuesta
	BufferedReader lectura = newBufferedReader(
			new InputStreamReader(conexionLectura));
	//Inicializa OBD asignando el protocolo automatico
	enviarComando.write("ATSP0" + "\r");
	enviarComando.flush();

	try {
		Thread.sleep(1000);
	} catch (InterruptedException e) {

		e.printStackTrace();
	}
```

```
//No guardamos la respuesta de la asignacion del protocolo, puesto
que no es relevante
lectura.readLine();
lectura.readLine();
lectura.readLine();

while (true) {

	//Cogemos la fecha actual que sera el id para los datos
	Date f = newDate();
	String date = f.toLocaleString();

	/* presion atmosferica */
	enviarComando.write("0133" + "\r");
	enviarComando.flush();

	try {
		Thread.sleep(1000);
	} catch (InterruptedException e) {

		e.printStackTrace();

	}

	//Unicamente guardamos en la variable la tercera lectura,
	es el valor del dato, el resto son datos irrelevantes
	lectura.readLine();
	lectura.readLine();
	atm = lectura.readLine();
	atm = atm.substring(6,8);
	atm_i = Integer.parseInt(atm, 16);
	atm_i=atm_i*10;

	/* temperatura ambiente */
	enviarComando.write("0146" + "\r");
	enviarComando.flush();

	try {
		Thread.sleep(1000);
	} catch (InterruptedException e) {

		e.printStackTrace();

	}

	lectura.readLine();
	lectura.readLine();
	temp = lectura.readLine();
	temp = temp.substring(6,8);
	temp_i = Integer.parseInt(temp, 16);
	temp_i=temp_i-40;


	/* temperatura aire entrada */
	enviarComando.write("010F" + "\r");
	enviarComando.flush();

	try {
		Thread.sleep(1000);
	} catch (InterruptedException e) {

		e.printStackTrace();

	}

	lectura.readLine();
	lectura.readLine();
	intemp = lectura.readLine();
```

```
            intemp = intemp.substring(6,8);
            intemp_i = Integer.parseInt(intemp, 16);
            intemp_i=intemp_i-40;

            /* temperatura refrigerante */
            enviarComando.write("0105" + "\r");
            enviarComando.flush();

            try {
                   Thread.sleep(1000);
            } catch (InterruptedException e) {

                   e.printStackTrace();

            }

            lectura.readLine();
            lectura.readLine();
            cooltemp = lectura.readLine();
            cooltemp = cooltemp.substring(6,8);
            cooltemp_i = Integer.parseInt(cooltemp, 16);
            cooltemp_i = cooltemp_i-40;

            /* tiempo motor encendido */
            enviarComando.write("011F" + "\r");
            enviarComando.flush();

            try {
                   Thread.sleep(1000);
            } catch (InterruptedException e) {

                   e.printStackTrace();

            }

            lectura.readLine();
            lectura.readLine();
            starttime = lectura.readLine();
            starttime=starttime.replace(" ", "");
            starttime = starttime.substring(4,8);
            starttime_i = Integer.parseInt(starttime, 16);
            starttime_i=starttime_i/60;

            //Insertamos todo en la base de datos
            cliente.dbInsertarComandos(date, atm_i, temp_i, intemp_i,
                                                  cooltemp_i, starttime_i);

            }
      }
```

Como se puede observar en el código anterior, una vez obtenidos todos los datos, se llama al método encargado de almacenar los datos en el servidor. Esto se explicará en el siguiente apartado.

5.3.1.2 Almacenando los datos

Es la segunda de las clases de esta parte del proyecto, y es la encargada de conectarse con la base de datos y enviar los datos recogidos al servidor. A continuación se expone el código encargado de realizar esta función:

```
public class ConnectDB{
Connection conexion = null;

	publicboolean dbConectar() {

		System.out.println("---dbConectar---");

		String driver = "org.postgresql.Driver";
		String servidor = "cardataobd.ddns.net"; // Direccion IP
		String puerto = "5432";
		String database = "OBDDatabase";
		String url = "jdbc:postgresql://" + servidor + ":" + puerto + "/"
				+ database;
		String user = "postgres";
		String password = "12345";

		try {

			System.out.println("---Conectando a PostgreSQL---");
			Class.forName(driver);
			conexion = DriverManager.getConnection(url, user,
password);
			System.out.println("Conexion realizada a la base de
datos");
			returntrue;

		} catch (ClassNotFoundException e) {
			e.printStackTrace();
			returnfalse;
		} catch (SQLException e) {
			e.printStackTrace();
			returnfalse;
		}
	}



	publicvoid dbInsertarComandos(String date, int atm, int temp, int
intemp, int cooltemp, int starttime) {
		Statement ps;
		String consulta;
		int resultado;
		try {
			ps = conexion.createStatement();
			consulta ="INSERT INTO obd VALUES ('"+date+"', '"+atm+"',
'"+temp+"','"+intemp+"', '"+cooltemp+"', '"+starttime+"')";
			resultado = ps.executeUpdate(consulta);
		} catch (SQLException e) {
			e.printStackTrace();
		}
	}

}
```

Como se puede observar el código es muy sencillo: se tienen dos módulos, el primero de ellos es el encargado de realizar la conexión con la base de datos *PostgreSQL* alojada en el servidor, y el segundo de ellos almacena sus parámetros de entrada, en dicha base de datos.

5.3.2 Scripts de los sensores añadidos

En esta segunda fase, se llevó a cabo el desarrollo de la parte encargada de almacenar y sacar partido a los datos recogidos por los sensores acoplados a la Raspberry Pi. Para ello se realizaron varios *scripts*[28] en *Python* encargados realizar estas labores.

Se tienen dos *scripts* principales, uno para cada sensor. Cada *script* obtiene de forma continua, los datos que recoge cada sensor, y realiza diversas labores con estos datos como se puede ver a continuación.

5.3.2.1 Sensor de temperatura

El *script* que se encarga del sensor de temperatura es bastante simple. Va recogiendo, con una cierta frecuencia, el valor de la temperatura devuelto por el sensor DHT11 y tras esto almacena en la base de datos del servidor el valor recogido. Por último, comprueba si el valor supera un cierto límite. De ser así, envía un aviso al *Telegram* del usuario. Veamos el código del *script*:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys
import time
import Adafruit_DHT
import os
import psycopg2

sensor = Adafruit_DHT.DHT11
pin = 4
update_0 = 0
update_1 = 0


while True:

temperature = Adafruit_DHT.read_retry(sensor, pin)

if temperature is not None:

        #Aqui cojo solo la parte de la temperatura, puesto que la primera parte
de la cadena es la humedad

        temp=format(temperature)[6:11]

        #Script que inserta la temperatura
        os.system("sudo python /home/pi/Desktop/TFG/BD/insertTemp.py "+ temp)

        print temp

        if float(temp)>40:

                #Script que manda aviso al Telegram
        os.system("sudo python /home/pi/Desktop/TFG/scripts/enviarmensaje.py
\"ATENCION!!! La temperatura en el interior del coche es superior a 40C\"")

        time.sleep(60)
```

A continuación, se verán los scripts utilizados a su vez en el *script* anterior. Primeramente, veremos el encargado de insertar los datos en la base de datos:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import psycopg2
import sys
import time

con = None

try:

con = psycopg2.connect("dbname='OBDDatabase' user='postgres'
host='cardataobd.ddns.net' port='5432' password='12345'")

cur = con.cursor()

cur.execute("INSERT INTO temp VALUES('"+time.strftime("%d-%b-%Y
%X")+"','"+sys.argv[1]+"')")

con.commit()


except psycopg2.DatabaseError, e:

if con:
con.rollback()

print 'Error %s' % e
sys.exit(1)


finally:

if con:
con.close()
```

Como se puede ver el código no tiene pérdida, conecta con la base de datos del servidor y, posteriormente, inserta la hora y fecha actual, junto con el valor que se le pasará como argumento.

Pasemos a ver ahora el otro *script* que se ha utilizado, en el *script* principal del sensor de temperatura. Su labor es, como ya sabemos, la de enviar un mensaje a la cuenta *Telegram* del usuario. A continuación se verá también el código de este:

```
#!usr/bin/env
import pexpect
import time
import sys

contacto = "Virgi"
mensaje = sys.argv[1]              #El mensaje será el primer argumento
telegram = pexpect.spawn('/home/pi/Desktop/TFG/tg/bin/telegram-cli -k
/home/pi/Desktop/TFG/tg/tg-server.pub') #Inicia Telegram
time.sleep(5)
telegram.sendline('contact_list')
telegram.sendline('msg '+contacto+' '+mensaje)   #Enviamos el msj al contacto
telegram.sendline('quit')                        #Cerramos Telegram
```

En el código anterior se está trabajando con el cliente de *Telegram* que se mencionó apartados anteriores. Este *script* hace lo siguiente, primero de todo se le indica el contacto al que se debe enviar el mensaje, luego el mensaje será el argumento que se le pase. Tras esto se ejecuta el cliente, y se carga la lista de contactos (esto es necesario para que reconozca todos los contactos almacenados en la cuenta), finalmente utilizando el comando "*msg*", perteneciente a la propia API de *Telegram*, y se envía el mensaje al contacto.

5.3.2.2 Sensor de monóxido de carbono

El *script* encargado de manejar los datos del sensor, es prácticamente idéntico al *script* de la temperatura. Los pasos que sigue son los mismos, recoge con una cierta frecuencia los datos del sensor de CO (únicamente nos interesa si el nivel es o no estable), y los almacena. En el caso de que el nivel de CO en el interior del coche no sea estable, enviará un mensaje al *Telegram*, al igual que sucedía con el *script* anterior. Aquí se puede observar el código:

```
import time, sys
import RPi.GPIO as GPIO
import os


GPIO.setmode(GPIO.BOARD)
GPIO.setup(7, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

#Se detecta CO alto en el ambiente
def action(pin):
    #Envia aviso al Telegram
os.system("sudo python /home/pi/Desktop/TFG/scripts/enviarmensaje.py
\"ATENCION!!! Altos niveles de CO2 en el interior del coche\"")
    #Almacena en la base de datos el nivel elevado
os.system("sudo python /home/pi/Desktop/TFG/BD/insertCo2.py " + "ELEVADO")
time.sleep(10)
return

GPIO.add_event_detect(7, GPIO.RISING)
GPIO.add_event_callback(7, action)

try:
while True:
os.system("sudo python /home/pi/Desktop/TFG/BD/insertCo2.py " + "ADECUADO")
time.sleep(10)
except KeyboardInterrupt:
GPIO.cleanup()
sys.exit()
```

Como se puede ver hay dos posibilidades: la primera de todas es si se activa el pin de la Raspberry Pi, esto quiere decir que hay una alta concentración de CO. En ese caso envía un aviso a la cuenta *Telegram* del usuario, y almacena el dato en la base de datos. La otra posibilidad es que no se active este pin, por lo que el nivel de CO sería estable. En este caso, únicamente se almacenaría el dato en la base de datos.

Se verán ahora los *scripts* utilizados dentro de este *script*. El primero que se verá es el encargado de enviar el aviso al *Telegram*. Se trata del mismo *script* que para el caso de la temperatura, por lo que no se va a volver a ver el código de este. El siguiente *script* que se

observará es el encargado de insertar el valor obtenido en la base de datos. El código del mismo es el siguiente:

```
import psycopg2
import sys
import time

con = None

try:

con = psycopg2.connect("dbname='OBDDatabase' user='postgres'
host='cardataobd.ddns.net' port='5432' password='12345'")

cur = con.cursor()

cur.execute("INSERT INTO gas VALUES('"+time.strftime("%d-%b-%Y
%X")+"','"+sys.argv[1]+"')")

con.commit()


except psycopg2.DatabaseError, e:

if con:
con.rollback()

print 'Error %s' % e
sys.exit(1)


finally:

if con:
con.close()
```

Como se puede ver, el código anterior, es casi idéntico al del *script* encargado de insertar en la base de datos la temperatura, lo único que varía en este caso, es la tabla origen en la que se insertará el dato.

5.3.3 Aplicación móvil

El desarrollo de la aplicación móvil conforma la tercera y última parte del desarrollo total del proyecto. En esta última fase, es donde se va a sacar provecho a todo el trabajo realizado anteriormente. Básicamente, consiste en una aplicación con la que poder consultar todos los datos que se han ido almacenando con anterioridad por diversos medios, ya sea a partir del coche o a partir de los sensores incorporados al mismo.

Primero de todo, se comenzó a trabajar con la pantalla principal, partiendo de la idea inicial de que debían distinguirse bien en ella tres partes diferentes, correspondientes a las distintas clases de datos que se iban a consultar. Se crearon para ello tres *Fragments* distintos, uno destinado a abarcar aquellos datos que pudieran influir en el estado del motor, otro

relacionado con datos del ambiente exterior, y, un último, que contiene los datos recogidos a partir de los sensores.

A continuación se puede observar a modo de ejemplo el código de uno de estos *Fragments*, concretamente el que hace referencia a los datos que pudieran influir en el estado del motor:

```
public class FragmentOBD extends Fragment {

    private Button btnInAirTemp, btnTimeStart, btnCoolTemp;

    public static FragmentOBD newInstance() {
        FragmentOBD fragment = new FragmentOBD();
        return fragment;
    }

    public FragmentOBD() {

    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_obd, container, false);



        btnCoolTemp = (Button) v.findViewById(R.id.button_CoolTemp);
        btnCoolTemp.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                //Creamos el Intent
                Intent intent = new Intent(getActivity(),
CoolTempActivity.class);

                //Iniciamos la nueva actividad
                startActivity(intent);
            }
        });

        btnInAirTemp = (Button) v.findViewById(R.id.button_AirTemp);
        btnInAirTemp.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

                Intent intent = new Intent(getActivity(),
InAirActivity.class);


                startActivity(intent);
            }
        });

        btnTimeStart = (Button) v.findViewById(R.id.button_TimeStart);
        btnTimeStart.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

                Intent intent = new Intent(getActivity(),
TimeStartActivity.class);
```

```java
                startActivity(intent);
            }
        });

        return v;
    }
```

Como se puede ver, el código de este *Fragment* es bastante simple, prácticamente lo único que contiene son las definiciones de los botones, así como el enlace de los mismos con su correspondiente redirección.

Los tres *Fragments* necesitan ser "manejados" por un adaptador. Para ello, se creó una nueva clase bastante simple que funciona como adaptador para la página de los *Fragments*. Éste es su código:

```java
public class MiFragmentPagerAdapter extends FragmentPagerAdapter {
    final int PAGE_COUNT = 3;
    private String tabTitles[] =
            new String[] { "Sensores","Motor", "Ambiente"};

    public MiFragmentPagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override
    public int getCount() {
        return PAGE_COUNT;
    }

    @Override
    public Fragment getItem(int position) {

        Fragment f = null;

        switch(position) {
            case 0:
                f = FragmentSensor.newInstance();
                break;
            case 1:
                f = FragmentOBD.newInstance();
                break;
            case 2:
                f = FragmentOBDAmb.newInstance();
                break;
        }

        return f;
    }

    @Override
    public CharSequence getPageTitle(int position) {
        // Generate title based on item position
        return tabTitles[position];
    }
}
```

Una vez hecho esto, se comenzó a trabajar con las pantallas correspondientes a la siguiente vista, aquellas, a las que se redirige la aplicación al pulsar alguno de los botones correspondientes al dato que se desea consultar. Es decir, las pantallas encargadas de mostrar

el valor del dato actual, así como la posibilidad de acceder al historial de este dato para una fecha concreta. A continuación se puede ver el código de una de estas vistas, concretamente se ha elegido para este caso la clase encargada de consultar la temperatura exterior:

```
Public class ExtTempActivity  extends AppCompatActivity {

    private TextView txtTemp;
    private Button btnBuscar;
    private Spinner spinnMes, spinnDia, spinnAno, spinnHora ;
    private String mon, year, hour, day;

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

        Toolbar toolbar = (Toolbar) findViewById(R.id.appbar);
        setSupportActionBar(toolbar);

        txtTemp = (TextView)findViewById(R.id.textExtTemp);
        new FetchSQL().execute();

        spinnMes = (Spinner)findViewById(R.id.Spinn_mes);
        spinnDia = (Spinner)findViewById(R.id.Spinn_dia);
        spinnAno = (Spinner)findViewById(R.id.Spinn_ano);
        spinnHora = (Spinner)findViewById(R.id.Spinn_hora);

        ArrayAdapter<CharSequence> adaptadorMes =
ArrayAdapter.createFromResource(this, R.array.valores_mes,
android.R.layout.simple_spinner_item);

        adaptadorMes.setDropDownViewResource(
                android.R.layout.simple_spinner_dropdown_item);

        spinnMes.setAdapter(adaptadorMes);

        spinnMes.setOnItemSelectedListener(
                new AdapterView.OnItemSelectedListener() {
                    public void onItemSelected(AdapterView<?> parent,
                                               android.view.View v, int
position, long id) {
                        mon = parent.getItemAtPosition(position).toString();
                    }

                    public void onNothingSelected(AdapterView<?> parent) {

                    }
                });

        ArrayAdapter<CharSequence> adaptadorDia =
ArrayAdapter.createFromResource(this, R.array.valores_dia,
android.R.layout.simple_spinner_item);

        adaptadorDia.setDropDownViewResource(
                android.R.layout.simple_spinner_dropdown_item);

        spinnDia.setAdapter(adaptadorDia);

        spinnDia.setOnItemSelectedListener(
                new AdapterView.OnItemSelectedListener() {
                    public void onItemSelected(AdapterView<?> parent,
                                               android.view.View v, int
position, long id) {
                        day = parent.getItemAtPosition(position).toString();
                    }

                    public void onNothingSelected(AdapterView<?> parent) {
```

```
                    }
                });

        ArrayAdapter<CharSequence> adaptadorAno =
ArrayAdapter.createFromResource(this, R.array.valores_ano,
android.R.layout.simple_spinner_item);

        adaptadorAno.setDropDownViewResource(
                android.R.layout.simple_spinner_dropdown_item);

        spinnAno.setAdapter(adaptadorAno);

        spinnAno.setOnItemSelectedListener(
                new AdapterView.OnItemSelectedListener() {
                    public void onItemSelected(AdapterView<?> parent,
                                               android.view.View v, int
position, long id) {
                        year = parent.getItemAtPosition(position).toString();
                    }

                    public void onNothingSelected(AdapterView<?> parent) {

                    }
                });

        ArrayAdapter<CharSequence> adaptadorHora =
ArrayAdapter.createFromResource(this, R.array.valores_hora,
android.R.layout.simple_spinner_item);

        adaptadorHora.setDropDownViewResource(
                android.R.layout.simple_spinner_dropdown_item);

        spinnHora.setAdapter(adaptadorHora);

        spinnHora.setOnItemSelectedListener(
                new AdapterView.OnItemSelectedListener() {
                    public void onItemSelected(AdapterView<?> parent,
                                               android.view.View v, int
position, long id) {
                        hour = parent.getItemAtPosition(position).toString();
                    }

                    public void onNothingSelected(AdapterView<?> parent) {

                    }
                });

        btnBuscar = (Button)findViewById(R.id.buttonBuscar);

        btnBuscar.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                //Creamos el Intent
                Intent intent =
                        new Intent(ExtTempActivity.this,
ExtTempRegisterActivity.class);

                //Creamos la informacion a pasar entre actividades
                Bundle bundle = new Bundle();
                bundle.putString("MES", mon.toString());
                bundle.putString("DIA", day.toString());
                bundle.putString("ANO", year.toString());
                bundle.putString("HORA", hour.toString());

                //Anadimos la informacion al intent
                intent.putExtras(bundle);
```

```
                //Iniciamos la nueva actividad
                startActivity(intent);
            }
        });

    }

    private class FetchSQL extends AsyncTask<Void,Void,String> {
        @Override
        protected String doInBackground(Void... Param) {
            String retval = "";
            try {
                Class.forName("org.postgresql.Driver");
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
                retval = e.toString();
            }
            String servidor = "cardataobd.ddns.net"; // Direccion IP
            String puerto = "5432";
            String database = "OBDDatabase";
            String url = "jdbc:postgresql://" + servidor + ":" + puerto + "/"
                    + database;
            String user = "postgres";
            String password = "12345";
            Connection conn;
            try {
                DriverManager.setLoginTimeout(5);
                conn = DriverManager.getConnection(url, user, password);
                Statement st = conn.createStatement();
                String sql;
                sql = "SELECT out_temp FROM obd ORDER BY date_id DESC LIMIT
1";

                ResultSet rs = st.executeQuery(sql);
                while(rs.next()) {
                    retval = rs.getString(1);
                }
                rs.close();
                st.close();
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
                retval = e.toString();
            }
            return retval;
        }

        @Override
        protected void onPostExecute(String value) {

            txtTemp.setText(value + " C");
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is
present.
getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();
```

```
        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            Intent op =
                    new Intent(ExtTempActivity.this, OptionsActivity.class);
            startActivity(op);
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

Como se puede, ver se trata de una clase bastante extensa, en la que hay muchas cosas que comentar. Se explicarán por lo tanto las partes importantes del código:

Al principio se puede observar la definición de varios *spinners*[29], y tras éstas sus correspondientes adaptadores. La función de estos *spinners*, no es otra que la de permitir al usuario elegir una fecha y hora concreta para consultar el historial de valores de este dato, en esta caso de la temperatura exterior. Posteriormente utilizando un *Bundle*[30], se pasa esta información a la siguiente actividad.

Justo después de esto, se encuentra una función que hereda de *AsyncTask*[31]. Como se puede observar esta función es la encargada de trabajar con la base de datos del servidor. Lo primero que hace, es conectar con el servidor para, posteriormente, realizar una consulta SQL a la base de datos deseada, con el objetivo de obtener el último dato insertado en la tabla, es decir, lo que sería el valor actual, finalmente se tiene el módulo *onPostExecute*, en el cual se obtiene el resultado de esta consulta.

El resto de clases que tienen la misma función que la anterior, pero consultando otro tipo de dato, son exactamente iguales, por lo que no es necesario volver a ver el mismo código.

Queda por ver únicamente, lo que sería el último nivel de pantallas de la aplicación, es decir, la pantalla a la que redirige la pantalla vista anteriormente, cuando se quiere consultar el historial del dato para una fecha concreta. Se verá a continuación el código que tiene la actividad encargada de esto. Como antes se ha visto como ejemplo la actividad encargada de la temperatura, se seguirá con ello, y se verá la actividad encargada de mostrar el historial de la temperatura:

```
public class ExtTempRegisterActivity extends AppCompatActivity {

    Date f = new Date();
    String date = f.toLocaleString();


    private String fecha, fechaFin;
    private TextView registro, noDatos;

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

        Toolbar toolbar = (Toolbar) findViewById(R.id.appbar);
```

```
        setSupportActionBar(toolbar);

        registro=(TextView)findViewById(R.id.textRegister);
        noDatos=(TextView)findViewById(R.id.textNoDatos);

        new FetchSQL().execute();


        //Recuperamos la informacion pasada en el intent
        Bundle bundle = this.getIntent().getExtras();
        fecha = bundle.getString("DIA") + "-" + bundle.getString("MES") + "-"
+ bundle.getString("ANO") + " " + bundle.getString("HORA") + ":00:00";
        fechaFin = bundle.getString("DIA") + "-" + bundle.getString("MES") +
"-" + bundle.getString("ANO") + " " + bundle.getString("HORA") + ":59:59";

    }

    private class FetchSQL extends AsyncTask<Void,Void,String> {
        @Override
        protected String doInBackground(Void... Param) {
            String retval = "";
            try {
                Class.forName("org.postgresql.Driver");
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
                retval = e.toString();
            }
            String servidor = "cardataobd.ddns.net"; // Direccion IP
            String puerto = "5432";
            String database = "OBDDatabase";
            String url = "jdbc:postgresql://" + servidor + ":" + puerto + "/"
                    + database;
            String user = "postgres";
            String password = "12345";
            Connection conn;
            try {
                DriverManager.setLoginTimeout(5);
                conn = DriverManager.getConnection(url, user, password);
                Statement st = conn.createStatement();
                String sql;
                sql = "SELECT date_id, out_temp FROM obd WHERE date_id BETWEEN
'"+fecha+"' AND '"+fechaFin+"'";
                ResultSet rs = st.executeQuery(sql);
                while(rs.next()) {
                    retval = retval + rs.getString(1) + " => "+
rs.getString(2) +" C"+ "\n";
                }
                rs.close();
                st.close();
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
                retval = e.toString();
            }
            return retval;
        }

        @Override
        protected void onPostExecute(String value) {

            if(value.isEmpty())
                noDatos.setText("ERROR: No hay datos registrados para esa
fecha");

            else
                registro.setText(value);

        }
```

```
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is
present.
getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {

        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            Intent op =
                    new Intent(ExtTempRegisterActivity.this,
OptionsActivity.class);
            startActivity(op);
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

Lo primero relevante que se puede observar en el código, es cuando se recupera la información pasada de la actividad anterior, esto se hace mediante un *Bundle*, obteniendo así la hora y fecha para ajustar la búsqueda. Lo siguiente que se observa es la función encargada de trabajar con la base de datos, al igual que en la actividad anterior. Ésta, del mismo modo que antes, conecta con el servidor y la base de datos concreta para, posteriormente realizar una consulta SQL, en la que obtiene los valores de todos los datos obtenidos durante la hora y fecha indicada. Posteriormente estos valores son recuperados en el método *onPostExecute*.

Al igual que lo que sucede con la actividad anterior, el resto de actividades encargadas de recuperar el historial de los otros datos, son prácticamente iguales, por lo que no es necesario volver a explicar el código.

Con todo lo que se ha visto anteriormente se puede formar una idea de cuál es el funcionamiento de la aplicación y cuáles son sus funcionalidades principales.

5.4 Pruebas

Una vez acabada la aplicación, ya estaba todo a punto para probar todo el proyecto en su conjunto. Se creó un *.jar*[32] de la aplicación Java encargada de conectarse y obtener datos del dispositivo acoplado al coche, se metió este *.jar* en la Raspberry Pi, y se acopló ésta al coche junto con los sensores. Tras esto se generó un *apk*[33] de la aplicación Android, y se instaló en el terminal móvil. Por último, se levantó el servidor que contiene la base de datos para, finalmente, probar la aplicación con todo el entramado funcionando. Se comprobó toda la funcionalidad de la aplicación y se observó que trabajaba de forma correcta.

Capítulo 6

Resultados y discusión

6. Resultados y discusión

En este apartado se verá todo lo relacionado con los datos recogidos y almacenados en la base de datos, a los cuales accede la aplicación móvil. Así mismo, se verá también alguna gráfica en la que se pueden observar la variación de los datos con respecto al tiempo, lo que es bastante útil puesto que si se viera algún pico muy pronunciado podría ser indicador de que algo no va bien.

Al igual que en los casos anteriores, los datos son divididos en dos grupos: uno primero en el que se verán todos los datos recogidos directamente del coche y otro correspondiente a los dos sensores acoplados a la Raspberry Pi.

6.1 Datos obtenidos del coche

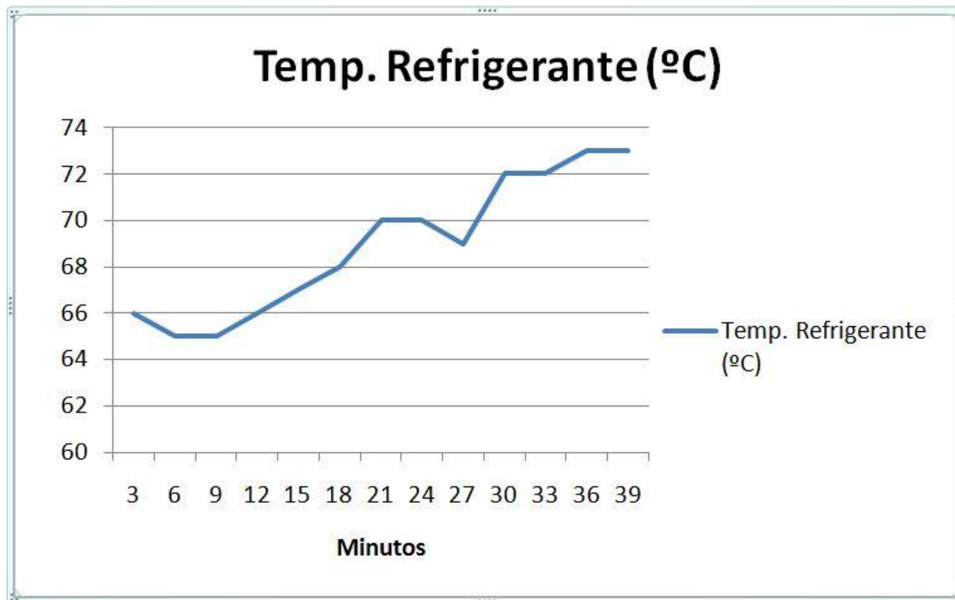
En la figura 6.1, se puede ver una porción de la tabla de los datos obtenidos del coche, los cuales fueron obtenidos durante el periodo de tiempo de una hora.

date_id [PK] character varying(40)	pressure character varying(4	out_temp character vary	in_air_temp character varying	coolant_temp character varyin	time_start character varying(4
22-jun-2015 23:00:02	97	32	38	66	575
22-jun-2015 23:03:28	97	32	39	65	781
22-jun-2015 23:06:33	97	32	38	65	966
22-jun-2015 23:09:35	97	32	39	66	1148
22-jun-2015 23:13:10	97	32	39	67	1363
22-jun-2015 23:16:15	97	32	39	68	1548
22-jun-2015 23:19:20	97	32	39	70	1733
22-jun-2015 23:22:25	97	32	39	70	1918
22-jun-2015 23:25:30	97	32	39	69	2103
22-jun-2015 23:28:35	97	32	39	72	2288
22-jun-2015 23:31:40	97	32	39	72	2473
22-jun-2015 23:34:45	97	32	37	73	2658
22-jun-2015 23:37:50	97	32	36	72	2843
22-jun-2015 23:40:55	97	32	36	73	3028
22-jun-2015 23:44:01	97	32	35	74	3214
22-jun-2015 23:47:06	97	32	35	74	3399
22-jun-2015 23:50:11	97	32	35	75	3584
22-jun-2015 23:56:17	97	32	35	75	3770
22-jun-2015 23:59:22	97	32	35	75	3955

Figura 6.1 Captura de la tabla en PgAdminIII

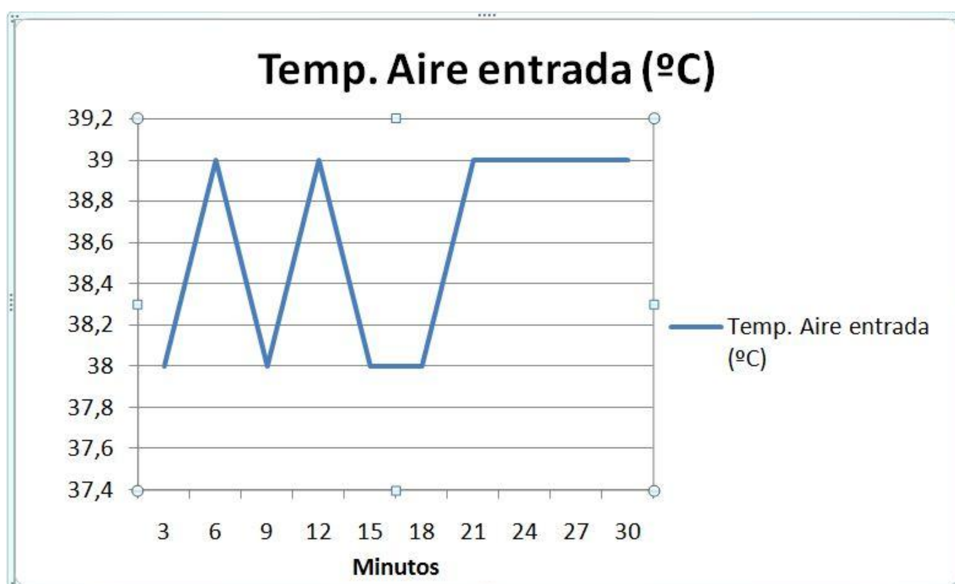
Como se puede observar en la tabla anterior, tanto la presión atmosférica como la temperatura exterior, se mantienen constantes, la temperatura del aire de entrada varía muy levemente, al igual que la temperatura del líquido refrigerante. En cambio, como es evidente, donde se puede ver más variación entre los valores, es en el tiempo del motor encendido. Todo esto se debe a que estos resultados se obtuvieron con el coche con el motor encendido pero estando estacionado.

A continuación se pueden ver unas gráficas para observar como varían los valores con respecto al tiempo. En la primera de ellas, se puede comprobar la variación sufrida por la temperatura, del líquido refrigerante del coche, en un periodo aproximado de 40 minutos:



Como se puede observar, en la gráfica hay dos subidas notables de los valores. Esto es debido a que se revolucionó el coche con el fin de ver cómo variaban los valores de temperatura. Tras esto, se dejó el coche sin revolucionar, y tras un tiempo la temperatura del refrigerante se mantuvo constante para, posteriormente, volver a bajar como se puede observar en la gráfica.

Se verá a continuación cómo varía la temperatura del aire de entrada al motor, con respecto al tiempo, en un periodo aproximado de media hora:



En este caso, a pesar de que los valores de la temperatura están variando casi continuamente, esta variación es ínfima, de un solo grado centígrado, por lo que no se podría sacar ninguna conclusión.

Graficas como las vistas anteriormente, son de gran utilidad, puesto que de haber existido algún dato anómalo que pudiera perjudicar al motor, en esta gráfica se podría haber descubierto con facilidad.

6.2 Datos obtenidos de sensores

En este apartado se verán los resultados obtenidos a partir de los sensores acoplados a la Raspberry Pi. Primeramente, se verán los datos obtenidos del sensor detector de monóxido de carbono. Esto se puede ver en la figura 6.2.

date_id [PK] character varying(40)	gas character varying(4(
08-Jul-2015 11:59:55	ADECUADO
08-Jul-2015 12:01:01	ADECUADO
08-Jul-2015 12:02:07	ADECUADO
08-Jul-2015 12:03:13	ADECUADO
08-Jul-2015 12:04:18	ADECUADO
08-Jul-2015 12:05:24	ADECUADO
08-Jul-2015 12:06:30	ADECUADO
08-Jul-2015 12:07:36	ADECUADO
08-Jul-2015 12:08:42	ADECUADO
08-Jul-2015 12:09:48	ADECUADO
08-Jul-2015 12:10:53	ADECUADO

Figura 6.2 Captura de la tabla en PgAdminIII

Como podemos observar en la tabla anterior, en todo momento los niveles de CO en el interior del coche se mantuvieron estables.

Queda ver por lo tanto, los datos obtenidos a partir del sensor de temperatura acoplado también a la Raspberry Pi. Los resultados obtenidos se pueden observar en la figura 6.3.

date_id [PK] character varying(40	temp character varying(40)
08-Jul-2015 12:48:13	29.0
08-Jul-2015 12:49:27	29.0
08-Jul-2015 12:50:33	29.0
08-Jul-2015 12:51:50	29.0
08-Jul-2015 12:52:56	29.0
08-Jul-2015 12:53:02	29.0
08-Jul-2015 12:54:11	29.0
08-Jul-2015 12:55:18	30.0
08-Jul-2015 12:56:24	30.0
08-Jul-2015 12:57:28	30.0

Figura 6.3 Captura de la tabla en PgAdminIII

Como se puede ver, en un periodo tan breve de tiempo, la temperatura en el interior del coche apenas sufrió cambios.

No se han realizado gráficas tanto del sensor detector de CO, como del el de temperatura, debido a que los valores, salvo caso inusual, se mantienen sin variación y, no es posible sacar conclusión alguna de los datos, o y no se visualizará cambio alguno que se pueda ver reflejado en una gráfica.

Capítulo 7
Conclusiones y líneas futuras

7. Conclusiones y líneas futuras

7.1 Conclusiones

En la última década, el desarrollo de los coches en el ámbito tecnológico está siendo abismal. La mayoría de los automóviles fabricados recientemente, por no decir la totalidad de estos, vienen diseñados para sincronizar el dispositivo móvil con el coche, vía *Bluetooth*. Quizás en una etapa no muy lejana, el móvil será imprescindible para poder conducir nuestro vehículo. De hecho, este mismo año la marca de automóviles BMW, ha diseñado una aplicación móvil, en la que se pueden ver datos del coche tales como el nivel de combustible, la autonomía, o el kilometraje[34].

Gracias a todo el proceso de investigación que se ha tenido que llevar a cabo acerca de este tema, para la realización del proyecto, así como los conocimientos que se han adquirido fruto de este trabajo, se puede decir que el móvil será una herramienta indispensable para los automóviles en un futuro no muy lejano, puesto que, la sociedad, está cada vez estamos más enganchada a éstos.

En este proyecto, se ha trabajado con herramientas que cualquiera puede obtener fácilmente. El dispositivo ELM327 mini, encargado de obtener información del coche, es muy básico, debido principalmente a su bajo coste. Y, a pesar de esto, se ha podido consultar datos de sensores que eran difíciles de saber de su existencia. Con unas herramientas más sofisticadas, se podrían realizar grandes avances y, permitir al usuario del vehículo, poder acceder a toda la información de su coche. Poder tener, como es en este caso, datos del motor en nuestro móvil, y poder consultarlos cuando nos venga en gana, puede hacer que se eviten muchas averías asociadas principalmente al mantenimiento del coche.

En conclusión, el resultado del proyecto ha sido satisfactorio. Como se ha dicho anteriormente, se han utilizado unas herramientas de bajo coste, cuyo consumo es muy bajo. Además todo el software utilizado durante el desarrollo del proyecto es libre, y aun así se ha podido obtener información relevante gracias a ellas. Además, la aplicación Android es útil e intuitiva, se puede acceder al valor actual del dato así como al historial del mismo, de forma rápida y sencilla. Por todo el trabajo que se ha realizado, así como por el resultado obtenido, se puede decir que el desenlace del proyecto ha sido para estar orgulloso de nosotros mismos.

7.2 Líneas futuras

A continuación se expondrán algunas de las ideas posibles que podrían ser implementadas en el caso de que se siguiera desarrollando el proyecto:

- Aumentar el número de tipos de datos obtenidos, para ello sería necesario adquirir un conector OBD de mayor calidad.

- Incluir la predicción de averías en la propia aplicación, para ello sería necesario recoger una cantidad de datos que se pudiera considerar significativa, y llevar a cabo un estudio exhaustivo de los mismos, para así poder sacar unos ciertos patrones, es decir, realizar con ellos un proceso de minería de datos.

- Dotar, de modo alguno, al sistema recolector de datos, de su propia conexión a internet, de tal manera que no dependa del dispositivo móvil.

- Que, cuando por algún tipo de problema, no se puedan enviar los datos al servidor, éstos se almacenen en la propia Raspberry Pi, para ser enviadas de forma automática cuando ésta disponga de conexión.

- Automatizar algunos mecanismos del coche, cuando se den unas circunstancias concretas, tales como, si en el interior del vehículo hace una temperatura superior a 40º y el motor está en marcha, se active de forma automática el climatizador.

Anexos

Anexo 1

Manual de usuario

Anexos

A.1 Manual de usuario

Lo primero que se ve cuando se ejecuta la aplicación, será la pantalla principal, en la cual se pueden ver 3 pestañas. Cada una de estas pestañas, hace referencia al tipo de dato que contiene, por defecto se abrirá la pestaña perteneciente a sensores, que hace referencia a los sensores acoplados a la Raspberry Pi.

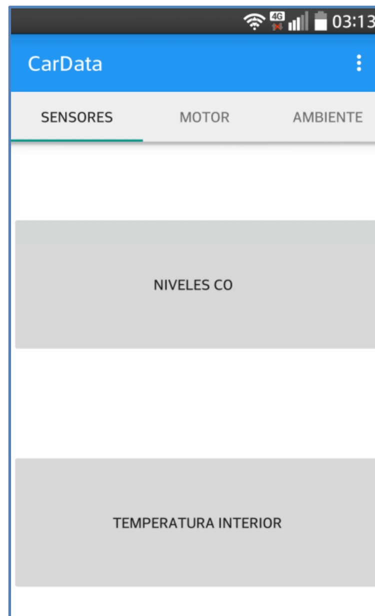


Figura A.1.1. Captura de la pantalla de sensores

Si se desliza un dedo sobre la pantalla, o bien se pincha sobre alguna de las pestañas, se podrá navegar por las diferentes páginas.

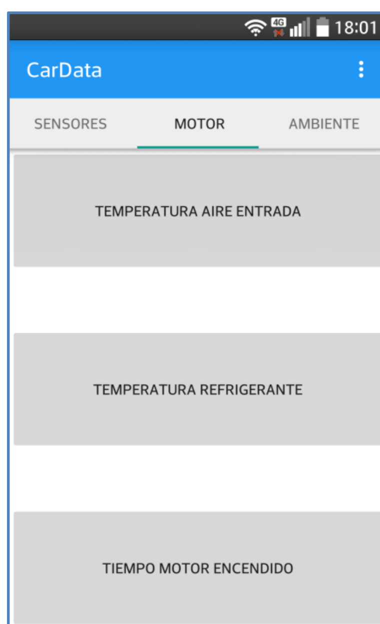


Figura A.1.2. Captura de la pantalla de motor

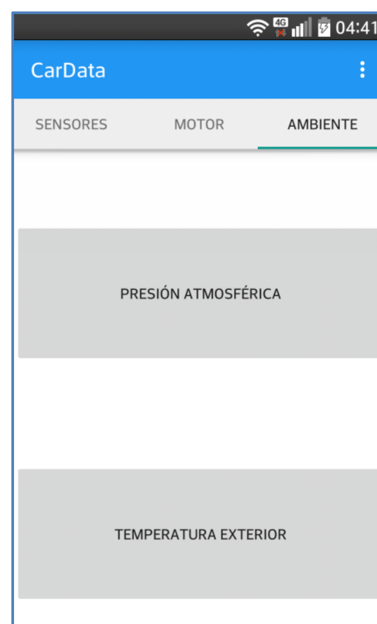


Figura A.1.3 Captura de la pantalla de ambiente

Estando en cualquiera de estas pantallas, únicamente se tiene que pulsar sobre el botón del dato que se desea consultar y, automáticamente, se redirigirá a la siguiente pantalla en la cual se puede ver el valor actual del dato, así como una opción que nos permitirá buscar todos los valores recogidos de ese dato para un hora y fecha en concreto. Se puede ver en la figura A.1.4, como ejemplo de esto, la pantalla que se mostraría al pinchar sobre la opción de "Temperatura exterior".

Figura A.1.4 Captura de la pantalla de consulta de la temperatura exterior

Una vez aquí se tiene la opción, como se ha dicho anteriormente, de consultar el historial de valores del dato en concreto. Bastaría con elegir la fecha y hora, pinchando para ello sobre las diferentes pestañas presentes en la pantalla, y se abrirá un desplegable para poder acotar la búsqueda, como se muestra en la figura A.1.5.

Figura A.1.5 Captura de consulta con la fecha desplegada

Una vez se tiene la hora y fecha elegida, bastaría con pulsar en el botón "Buscar", automáticamente aparecerá una nueva pantalla con los datos deseados.

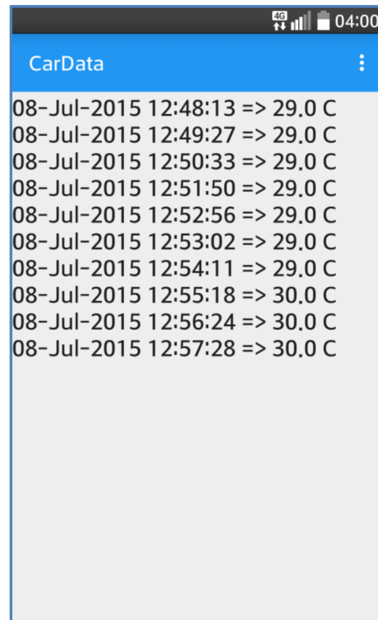


Figura A.1.6 Captura de la pantalla del historial de temperatura

En el caso de que por error, o por algún otro motivo no existan en la base de datos, datos almacenados para ese rango de fechas, saldrá un mensaje de error, como se puede observar en la figura A.1.7.

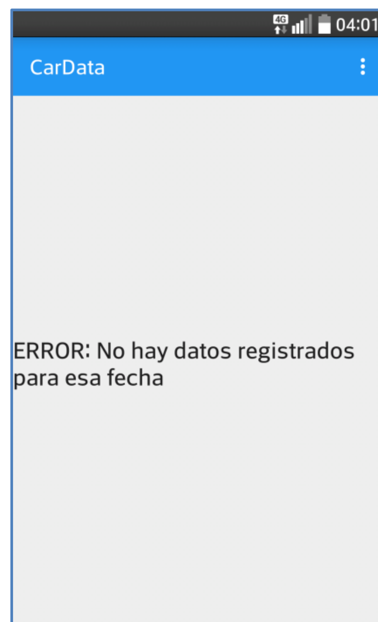


Figura A.1.7 Captura de la pantalla de aviso de error

Las demás pantallas que forman parte de los diferentes tipos de datos, se comportan de la misma manera que se ha explicado anteriormente, por lo que no es necesario explicar dichas pantallas.

Anexo 2

Casos de uso

A.2 Casos de uso

En este apartado, se verán tres pequeños diagramas de casos de uso, correspondientes a las acciones que puede hacer el usuario en la aplicación móvil.

El primer caso de uso que se va a ver, es el caso de uso correspondiente a la acción de consultar los datos recogidos por lo sensores (figura A.2.1).

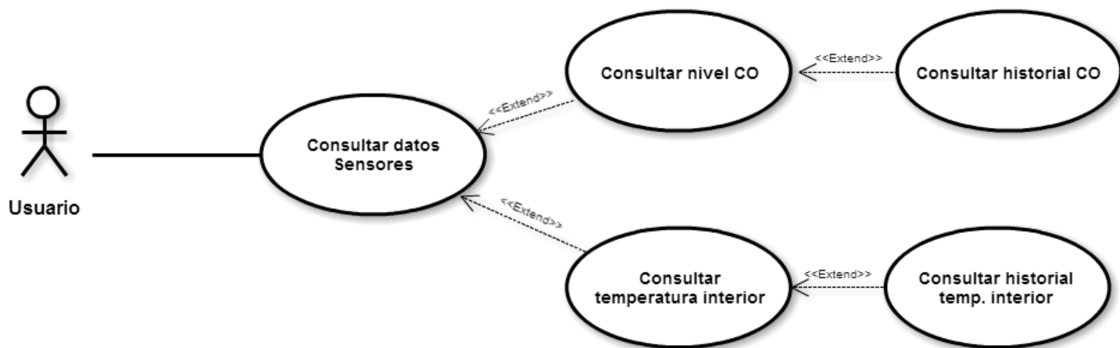


Figura A.2.1 Imagen del caso de uso de consultar datos sensores

El segundo caso de uso que se muestra, es el correspondiente a la acción de consultar los datos relevantes para el motor (figura A.2.2).

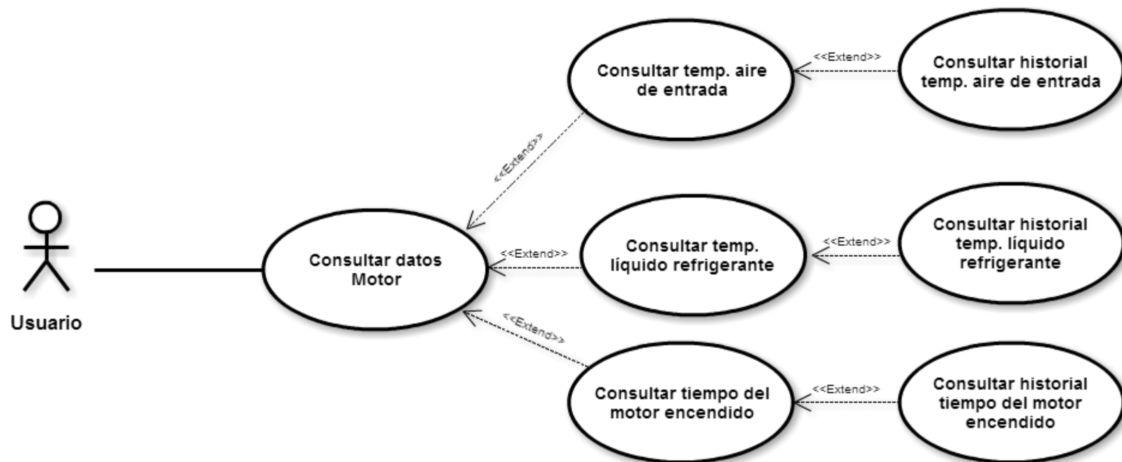


Figura A.2.2 Imagen del caso de uso de consultar datos del motor

El tercer y último caso de uso que se va a ver, es el encargado de representar la acción de consultar los datos relacionados con el ambiente (figura A.2.3).

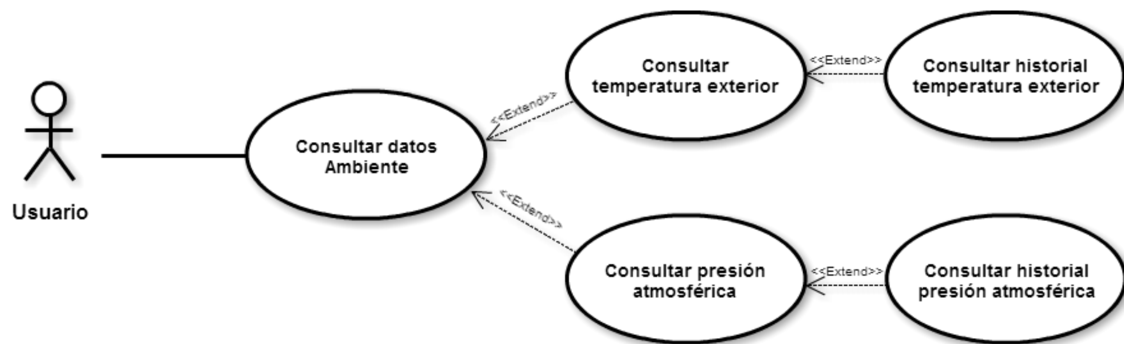


Figura A.2.3 Imagen del caso de uso de consultar datos del ambiente

Anexo 3

Glosario

A.3 Glosario

[1]Raspberry Pi: se trata de un ordenador de placa reducida, de un coste pequeño, y que fue creado en Reino Unido por la fundación Raspberry Pi, con el propósito de estimular la enseñanza de la informática en las escuelas.

[2]Telegram: servicio de mensajería por Internet basado en la nube.

[3]Wereables: dispositivos electrónicos que se acoplan a alguna parte de nuestro cuerpo, con el objetivo de interactuar con el usuario, u otros dispositivos con el objetivo de realizar alguna función específica.

[4]*Fuente*: http://www.ericsson.com/blogs/sociedad-conectada/2015/04/07/sociedad-conectada-debajo-de-la-piel-conozca-los-wearables-del-futuro/.

[5]*Fuente*: http://hipertextual.com/2014/12/pruebas-de-comunicacion-entre-coches.

[6]*Fuente*: https://es.wikipedia.org/wiki/OBD.

[7]*Fuente*: https://es.wikipedia.org/wiki/Bluetooth.

[8]*Fuente*: http://www.postgresql.org.es/sobre_postgresql.

[9]*Fuente*: https://es.wikipedia.org/wiki/Eclipse_(software).

[10]Plug-in: se trata de un complemento que se asocia con a una aplicación, para dotar a esta de una función nueva y, por lo general, muy específica.

[11]IntelIJ IDEA: se trata de un entorno de desarrollo integrado (IDE) para el desarrollo de programas informáticos.

[12]*Fuente*: https://es.wikipedia.org/wiki/Android_Studio.

[13]Gradle: es una herramienta que permite automatizar el desarrollo de los proyectos, en tareas tales como compilación, testing, empaquetado, y el despliegue de los mismo.

[14]Android Wear: sistema operativo para dispositivos *wereables*.

[15] Multiparadigma: se aplica a aquel lenguaje que soporta más de un paradigma de programación.

[16] Tipado dinámico: una misma variable puede tomar valores distintos en distintos momentos.

[17] *Fuente*: https://es.wikipedia.org/wiki/Python

[18]*Fuente*: https://telegram.org/.

[19]*Fuente*: https://appinformatica.com/portatiles-asus.-k550cc-xx509h-i7-3537u-8gb-1tb-vga-2-w8-15.6.php.

[20]*Fuente*: http://www.lg.com/es/telefonos-moviles/lg-G2-D802.

[21]*Fuente*: http://hipertextual.com/2014/07/raspberry-pi-b.

[22]*Fuente*: http://www.amazon.es/Herramienta-Bluetooth-Interfaz-diagn%C3%B3stico-explorador/dp/B00DY3EZSI

[23]*Fuente*: https://www.sparkfun.com/datasheets/Sensors/Biometric/MQ-7.pdf

[24]*Fuente*: http://issuu.com/rduinostar/docs/dht11_datasheet

[25]Fragment: se trata de un elemento de programación Android, que funciona dentro del ámbito de una *Activity* y, cuya función es la de ampliar parte de la lógica usada para la navegación entre *Activities*, pudiendo declarar varios Fragmentes dentro de una *Activity*.

[26]Toolbar: barra de herramientas de una aplicación Android.

[27]Main Activity: actividad principal de una aplicación Android.

[28]Script: se trata de un programa, que normalmente es simple, y que, por normal general, se guarda en texto plano.

[29]Spinner: elemento de desarrollo Android que, muestra una lista desplegable, para que el usuario seleccione un elemento de la misma.

[30]Bundle: se trata de un objeto de desarrollo Android que, permite pasar datos entre *Activities*.

[31]Asyntack: clase abstracta Android que permite realizar, de manera simple, tareas en segundo plano.

[32].jar: extensión perteneciente a los archivos que permiten ejecutar aplicaciones desarrolladas en Java.

[33].apk: archivo que permite distribuir e instalar componentes empaquetados para Android.

[34]*Fuente*: https://itunes.apple.com/es/app/my-bmw-remote/id387675830?mt=8.

Bibliografía

Using the Java APIs for Bluetooth Wireless Technology, Part 1 – API Overview

Disponible: http://www.oracle.com/technetwork/articles/javame/index-156193.html

Getting Started with Java and Bluetooth

Disponible: https://today.java.net/pub/a/today/2004/07/27/bluetooth.html

Javax.bluetooth Class LocalDevice

Disponible: http://bluecove.org/bluecove/apidocs/javax/bluetooth/LocalDevice.html

OBD-II PIDS

Disponible: https://en.wikipedia.org/wiki/OBD-II_PIDs

ELM327 AT Commands

Disponible: https://docs.oracle.com/javase/tutorial/jdbc/basics/

JDBC Tutorial

Disponible: https://geekytheory.com/tutorial-raspberry-pi-uso-de-telegram-desde-la-terminal/

Tutorial Raspberry Pi – Uso de Telegram desde la terminal

Disponible: https://geekytheory.com/tutorial-raspberry-pi-uso-de-telegram-desde-la-terminal/

Tutorial Raspberry Pi – Uso de Telegram con Python

Disponible: https://geekytheory.com/tutorial-raspberry-pi-uso-de-telegram-con-python/

Using the Adafruit BMP Python Library

Disponible: https://learn.adafruit.com/using-the-bmp085-with-raspberry-pi/using-the-adafruit-bmp-python-library

PostgreSQL Python Tutorial

Disponible: http://zetcode.com/db/postgresqlpythontutorial/

Android Studio Overview

Disponible: http://developer.android.com/tools/studio/index.html

Training for Android developers

Disponible: http://developer.android.com/training/index.html

Curso de programación Android

Disponible: http://www.sgoliver.net/blog/curso-de-programacion-android/indice-de-contenidos/

Android and PostgreSQL

Disponible: https://www.pgcon.org/2011/schedule/attachments/194_pgcon2011-pgdroid.pdf