UNIVERSIDAD DE ALMERÍA

ESCUELA POLITÉCNICA SUPERIOR Y FACULTAD DE CIENCIAS EXPERIMENTALES

INGENIERÍA INFORMÁTICA

DESARROLLO DE UN SÍSTEMA PARA EL ENVÍO Y RECEPCIÓN EN TIEMPO REAL DE INCIDENCIAS DE TRÁFICO UTILIZANDO GEOPOSICIONAMIENTO Y SERVICIOS WEB DE GOOGLE Y TWITTER

El Alumno:

Alejandro Pérez Manzano

Almería, 12 de junio de 2014

Director(es):

Dr. José Antonio Torres Arriaza

Agradecimientos

Dedico este proyecto a todos aquellos que me han apoyado y ayudado, haciendo posible la terminación del mismo.

Gracias a mi familia, que ha estado siempre conmigo y se ha asegurado de que llegue a donde estoy ahora.

Agradezco enormemente al Dr. José Antonio Torres Arriaza, por confiar en mí, y por ayudarme a llevar el proyecto por buen camino. Sin él no existiría este proyecto.

También agradezco al resto de profesores que han contribuido en mi formación en la Universidad. Gracias a ellos puedo decir que he aprendido lo necesario para convertirme en un buen profesional.

Y sobre todo, quiero agradecer a Aída, por ser un apoyo incondicional y ayudarme en todo momento.

Muchas gracias a todos. Os dedico este proyecto.

Prólogo

A día de hoy, los servicios en Internet están a la orden del día. Cada vez se utilizan más herramientas muy útiles basados en Internet, como el almacenamiento online, el geoposicionamiento, las redes sociales o servicios de streaming, entre multitud de otros servicios.

Por ello, es muy importante que estos servicios sean accesibles de forma fácil y sencilla, y que sean comunicables entre ellos. Esto ha propiciado el auge de los servicios web, que permiten a los desarrolladores de aplicaciones acceder a una gran cantidad de servicios de terceros, e integrarlos de forma relativamente sencilla dentro de sus aplicaciones, incrementando significativamente la utilidad y capacidad de sus aplicaciones, en beneficio de todos (desarrolladores, usuarios, plataformas, etc.)

Dentro de este panorama, los dispositivos inteligentes, o smartphone, han contribuido enormemente a la expansión y popularidad del uso de estos servicios y aplicaciones, además de abrir paso a muchas aplicaciones que se benefician de la movilidad, en especial aquellas que reaccionan a la localización del usuario en tiempo real.

El campo de aplicaciones basadas en servicios y geoposicionamiento es muy amplio, y por tanto se presta a realizar multitud de posibles proyectos y aplicaciones. El sistema que se decidió diseñar finalmente es un sistema de posicionamiento de incidencias de tráfico en tiempo real, mediante el cual enviamos la localización de una incidencia mediante los datos de GPS de un smartphone y luego podemos monitorizar en tiempo real los datos de las distintas incidencias conectándonos a un servidor que se encarga de presentar la información de dichas incidencias en un mapa de la ciudad, para mayor comodidad de los usuarios.

Es evidente la utilidad de un sistema como este, ya que podría conseguir una mejor gestión del tráfico dentro de la ciudad, mejorando su fluidez y evitando atascos o accidentes, por ejemplo. Además, existe una escasez de servicios similares para la gestión de incidencias de tráfico, con lo cual existe una necesidad de cubrir dicha escasez con un buen sistema que permita realizar dicha gestión, como el que propongo en este proyecto.

Índice

1. **Introducción** ... 1
1.1. Motivación ... 1
1.2. Objetivos ... 1
1.3. Metodología ... 2
2. **Servicios Web** ... 3
2.1. Introducción ... 3
2.2. Protocolos ... 3
2.2.1. Introducción ... 4
2.2.2. Transporte ... 5
2.2.3. Invocación ... 8
2.2.4. Descripción ... 22
2.2.5. Descubrimiento ... 27
2.3. Librerías de Twitter ... 28
2.3.1. Introducción ... 28
2.3.2. Autorización OAuth ... 30
2.3.3. API REST ... 32
2.3.4. API Streaming ... 34
2.4. Librerías de Google ... 36
2.4.1. Introducción ... 36
2.4.2. Google Maps ... 37
3. **Tecnología Android** ... 42
3.1. Introducción ... 42
3.2. Esquema básico de una aplicación ... 45
3.2.1. Actividades ... 45
3.2.2. Ciclo de vida ... 48
3.2.3. Recursos ... 50
3.2.4. Comunicación ... 52
3.2.5. Concurrencia ... 55
3.3. Acceso a la localización ... 60
3.3.1. Fuentes de localización ... 60
3.3.2. Refinamiento de la localización ... 63
4. **Desarrollo del trabajo** ... 65
4.1. Desarrollo del cliente ... 65

4.1.1. Introducción 65
4.1.2. Actividad principal 65
4.1.3. Servicio de localización 69
4.1.4. Autentificación en Twitter 71
4.1.5. Envío de datos 74
4.2. Desarrollo del servidor 77
4.2.1. Introducción 77
4.2.2. Arquitectura 79
4.2.3. Obtención de información 80
4.2.4. Persistencia 81
4.2.5. Diseño de la interfaz web 82
4.2.6. API de Google Maps 83
4.2.7. AJAX 84
5. **Conclusiones finales** 87
5.1. Objetivos alcanzados 87
5.2. Objetivos formativos alcanzados 87
5.3. Extensiones futuras del proyecto 88
6. **Bibliografía** 89
7. **Anexos** 93
7.1. Manual de uso 93
7.1.1. Instalación del cliente 93
7.1.2. Uso del cliente 98
7.1.3. Instalación del servidor 104
7.1.4. Uso del servidor 108
7.2. Diagramas de la aplicación 110
7.2.1. Casos de uso 110
7.2.2. Clases 113
7.2.3. Secuencia 115
7.2.4. Estados 116
7.3. Código fuente 117
7.3.1. Cliente 117
7.3.2. Servidor 166

Índice de ilustraciones

Ilustración 1 – Pila de protocolos de servicios web 4
Ilustración 2 – Comunicación en Web Services........ 5
Ilustración 3 – Estructura de un mensaje SOAP........ 13
Ilustración 4 – Comunicaciones en SOAP........ 18
Ilustración 5 – Estructura de un mensaje WSDL 24
Ilustración 6 – Esquema de autentificación OAuth........ 31
Ilustración 7 – Flujo de conexión en API REST 35
Ilustración 8 – Flujo de conexión en API Streaming........ 35
Ilustración 9 – Cuota de mercado de smartphones en 2010 43
Ilustración 10 – Cuota de mercado de smartphones en 2013 43
Ilustración 11 – Captura de pantalla de una Actividad en Android 46
Ilustración 12 – Ciclo de vida de una actividad en Android 49
Ilustración 13 – Actividad principal del cliente Android 66
Ilustración 14 – Actividad principal en inglés........ 68
Ilustración 15 – Actividad principal en modo landscape 69
Ilustración 16 – Página de desarrolladores de Twitter 71
Ilustración 17 – Autentificación OAuth en Twitter 72
Ilustración 18 – Respuesta de la petición OAuth de Twitter........ 73
Ilustración 19 – Captura de ejemplo de un tweet codificando una incidencia........ 75
Ilustración 20 – Incidencia en proceso de envío 76
Ilustración 21 – Incidencia enviada con éxito 77
Ilustración 22 – Ejemplo de la aplicación web 78
Ilustración 23 – Información detallada de una incidencia 78
Ilustración 24 – Esquema básica de la arquitectura del sistema 79
Ilustración 25 – Captura de ejemplo de la aplicación Web........ 82
Ilustración 26 – Exportación de la aplicación Android........ 93
Ilustración 27 – Creación de una clave de desarrollo Android 94
Ilustración 28 – Captura del gestor de archivos con el APK. 95
Ilustración 29 – Selección del APK........ 96
Ilustración 30 – Permisos de la aplicación Android 97
Ilustración 31 – Fin de la instalación 98
Ilustración 32 – Actividad principal 99
Ilustración 33 – Autentificación en Twitter (I) 100
Ilustración 34 – Autentificación en Twitter (II) 101
Ilustración 35 – Redirección a la aplicación 102
Ilustración 36 – Aplicación Android con Twitter activo 103
Ilustración 37 – Descarga de XAMPP 104
Ilustración 38 – Descarga de Apache Tomcat 105
Ilustración 39 – Página de inicio de Tomcat........ 106
Ilustración 40 – Configuraciones de usuarios en Tomcat 107
Ilustración 41 – Gestor de aplicaciones Web de Tomcat........ 108
Ilustración 42 – Servidor web del sistema de incidencias........ 109
Ilustración 43 – Detalles de una incidencia........ 109
Ilustración 44 – Diagrama de casos de uso de la aplicación. 111
Ilustración 45 – Diagrama de casos de uso del back-end 112

Ilustración 46 – Diagrama de casos de uso del front-end 113
Ilustración 47 – Diagrama de clases del cliente Android 113
Ilustración 48 – Diagrama de clases del back-end 114
Ilustración 49 – Diagrama de clases del front-end.. 114
Ilustración 50 – Diagrama de secuencia de la autentificación en Twitter del cliente.............. 115
Ilustración 51 – Diagrama de secuencia del envío de incidencias del cliente.......................... 115
Ilustración 52 – Diagrama de secuencia de la obtención de nuevas incidencias del servidor.. 116
Ilustración 53 – Diagrama de secuencia de la búsqueda de incidencias en el sistema 116
Ilustración 54 – Diagrama de estados del cliente Android 117
Ilustración 55 – Diagrama de estados del servidor (back-end).. 117

Índice de tablas

Tabla 1 – Operaciones REST 19
Tabla 2 – Fuentes de localización.. 60

1. Introducción

1.1. Motivación

Ya que a día de hoy estamos experimentando una gran expansión en el número y capacidades de los llamados teléfonos inteligentes (o smartphone) es lógico pensar en el aprovechamiento de estos recursos a la hora de crear aplicaciones útiles y novedosas. Ya que la gran ventaja que ofrece un smartphone respecto a un computador normal son tanto la movilidad (te lo puedes llevar a cualquier parte en el bolsillo) como el acceso a recursos varios tales como sensores o GPS, las posibilidades que se obtienen son por tanto impresionantes, y permiten realizar muchas aplicaciones de utilidad para ayudar a multitud de tareas cotidianas a todos esos usuarios de smartphone.

Basándonos en estas premisas es claro ver el potencial de una aplicación que permita utilizar los datos de posicionamiento geográfico del móvil. Se podrían desarrollar muchísimas aplicaciones a partir de la idea de la localización, y se hecho ya existen muchas de estas aplicaciones (suelen llamarse location-aware apps). Por tanto se partió de la base de realizar una tecnología lo más sencilla de usar posible para obtener y enviar estas localizaciones desde el móvil para poder utilizarse como base para muchas aplicaciones distintas.

Finalmente se determinó poder realizar una aplicación de incidencias de tráfico. Cualquiera que haya conducido en la ciudad sabe lo molesto que puede ser encontrarse con imprevistos tales como una calle cortada, un accidente o unas obras, entre otros. Cualquiera de estos sucesos perjudica a la fluidez del tráfico y a la capacidad de ajustar nuestras rutas y planificación del tiempo. Si pudiésemos avisar en tiempo real de estas incidencias junto a su localización atajaríamos estos problemas, y con una aceptación de la aplicación lo suficientemente amplia de la aplicación se ayudaría mucho en la conducción diaria de estos usuarios.

Con respecto al uso de servicios web sólo se puede decir que es evidente el gran impacto que tienen en la sociedad todos los servicios ofrecidos en Internet por grandes empresas como las ya omnipresentes redes sociales (Facebook, Twitter, LinkedIn,...), buscadores (Google, Bing,...), streaming de audio y video (Grooveshark, Spotify, YouTube,...) y muchos otros.

Estas empresas nos permiten acceder a todos esos usuarios, junto a los datos que se manejan (el Big Data) y todas las opciones que ofrecen en cuanto a conectividad de usuario. Esto se consigue gracias a que la mayoría de estos servicios disponen de APIs para su uso en aplicaciones de terceros, generalmente basadas en la tecnología de Servicios Web.

Es evidente entonces el potencial que pueden tener estas opciones para hacer el sistema desarrollado mucho más interesante de cara al usuario.

1.2. Objetivos

El objetivo final del proyecto es desarrollar un sistema completo para la gestión de incidencias de tráfico en tiempo real. Se pretende que el sistema se pueda utilizar en un entorno real y que tenga una utilidad final práctica.

Para ello el sistema necesita realizar una obtención de la localización en tiempo real basada en el GPS de nuestro smartphone para poder conformar la base de la información de la incidencia, que se completará con algunos datos adicionales aportados por el usuario.

Esta información por sí misma es bastante pobre y difícil de aprovechar, por tanto es necesario de la implementación de un mapa en el cual se posicionen mediante marcadores las distintas incidencias, para poder verlas de forma rápida y visual. Este mapa se mostrará a partir de un servidor Web, por lo cual se dispondrá de él en cualquier PC convencional sin más necesidad que una conexión a Internet.

Resumiendo, lo que se necesita es poder enviar los datos de las incidencias actuales de tráfico a partir de un smartphone con conexión a Internet y GPS hacia un servidor Web, que se encargará de recoger correctamente los datos y gestionarlos para poder presentarlos de forma sencilla en un mapa, junto con su información correspondiente.

1.3. Metodología

Para el desarrollo del proyecto me he basado en las metodologías ágiles, buscando un desarrollo iterativo en el cual, a cada iteración tenía siempre un producto completo, pero con alguna o algunas características más que la vez anterior, quedando así refinado en sucesivas iteraciones hasta llegar a un estado en el que se podía considerar el desarrollo como satisfactorio para cumplir los objetivos que se pretendían.

Con respecto a los requisitos tecnológicos del proyecto, se requiere de un smartphone Android 2.2 o superior para poder ejecutar la aplicación cliente y un servidor, para el cual será necesaria una capacidad proporcional a la carga de usuarios que deba soportar. La aplicación Web está debidamente optimizada y para cargas pequeñas de prueba bastará con un PC cualquiera de gama baja. Si quisiéramos poder escalarlo a un mayor número de usuarios, sería necesario irnos a un servidor dedicado, sobre todo especializado para un mayor número de transacciones.

En cuanto al software necesario para el servidor, debido a que el desarrollo de las librerías se ha hecho en Java, para que sean compatibles tanto para el cliente Android como para el servidor, será necesario un servidor basado en Java, que sea capaz de ejecutar páginas en JSP, en mi caso en particular lo he probado en un Apache Tomcat. También se requiere de una base de datos MySQL para la persistencia de los datos (incidencias) una vez recogidas de Internet.

2. Servicios Web

2.1. Introducción

Nos encontramos en un entorno en el cual la interconectividad entre múltiples sistemas informáticos es un problema cada vez más grande. Los distintos sistemas deben comunicar sus datos y resultados a otros sistemas y a sus usuarios. Esto hace que sea necesario un estándar que permita de forma sencilla realizar llamadas a las APIs de distintos sistemas, aunque sean muy heterogéneos y residan en distintas redes y bajo distintas organizaciones.

Tradicionalmente, para realizar computación distribuida entre distintos sistemas o servidores se han utilizado protocolos diseñados para el paso de mensajes entre sistemas distribuidos. Alguno de estos sistemas más utilizados podrían ser CORBA [1] o RMI [2]. El problema de estos sistemas es que requieren de una arquitectura específica, de manera que una aplicación o sistema con CORBA (como ejemplo) sólo será interoperativo con otro sistema que también se haya implementado siguiendo la misma arquitectura y especificación de CORBA. Además, también suelen tener un mayor coste de implementación y mantenimiento.

Como solución se desarrolló una nueva tecnología para la llamada a sistemas remotos denominada *Servicios Web*, o *Web Services*. Esta tecnología se basa en utilizar los estándares web ya existentes (como HTTP o XML) para encapsular la llamada a funciones o procedimientos de otro sistema, a través de Internet. Al usar protocolos y estándares abiertos ya asentados, implementados en multitud de servidores web, es fácil adaptar cualquier sistema para publicar su funcionalidad en base a servicios web [3].

Esta tecnología ha tenido en los últimos años un auge espectacular, convirtiéndose en la base de una de las arquitecturas software más importante de los últimos años, la arquitectura orientada a servicios (*Service Oriented Architecture, o SOA)*. Esto consiste en que un sistema completo se ve como una caja negra, en la cual se ofrecen una serie de servicios al exterior, que conformarán la interfaz de acceso al sistema. Así se podrán conectar componentes de forma sencilla para poder realizar sistemas más complejos con poco esfuerzo, o ampliar la funcionalidad de otros sistemas en base a servicios de terceros [4].

Por último, hay que destacar el hecho que la tecnología basada en servicios Web ha propiciado el aumento del *cloud computing* y los servicios través de Internet. A partir de este paradigma, se permite a los usuarios finales utilizar aplicaciones web y sistemas basados en servicios, directamente a través de un navegador Web, sin la necesidad de utilizar multitud de aplicaciones pesadas que requieran instalación en el cliente, mucho más engorroso y, sobre todo, con un mayor coste [5].

2.2. Protocolos

2.2.1. Introducción

Al estar basados en la comunicación Web, es importante que los protocolos que soportan los servicios Web sean de texto plano, y no binarios. Esto es así ya que el protocolo HTTP, que sirve de base para las comunicaciones cliente-servidor en Internet sólo permite comunicaciones de texto. Además, es importante que estos protocolos sean sencillos y abiertos, ya que deben poder ser implementados por cualquier API, en base a muchas tecnologías de programación diferentes (PHP, C#, Java, etc.).

Para la realización de las distintas etapas de la comunicación entre el cliente y el servidor mediante servicios web se utiliza una pila de protocolos, a la cual se la suele denominar “Web Services Protocol Stack”. Estos protocolos están definidos de forma estandarizada por la organización WS-I (Web Service Interoperability Organization) mediante los llamados perfiles [6]. Aunque se definen varios perfiles, los más importantes y usados son el básico y el de seguridad.

A efectos del proyecto me basaré en el perfil básico (WS-I Basic Profile v 2.0), ya que el de seguridad no hace más que añadir una capa de seguridad mediante el uso de WS-Security.

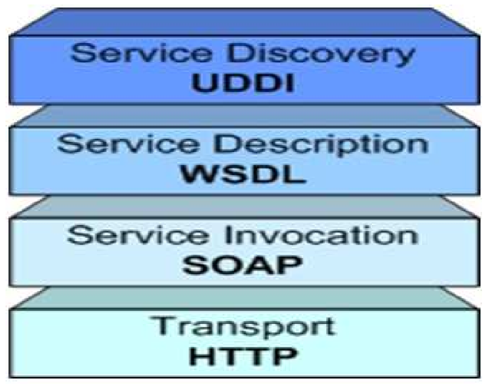


Ilustración 1 – Pila de protocolos de servicios web

Como se ve en el diagrama, tenemos cuatro capas principales a desarrollar en la comunicación:

- Transporte del servicio (HTTP y otros)
- Invocación del servicio (SOAP y REST)
- Descripción del servicio (WSDL)
- Descubrimiento de servicios (UDDI)

La manera en la que estas distintas capas se relacionan en una comunicación real se puede observar en el siguiente diagrama:

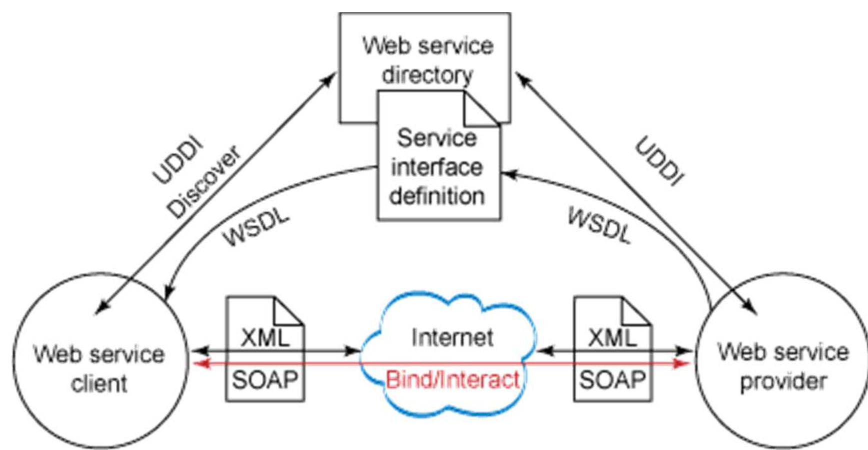


Ilustración 2 – Comunicación en Web Services

Explicaré el diagrama de forma resumida, ya que las tecnologías utilizadas serán explicadas en sus respectivos apartados.

El cliente, que quiere consumir un servicio web de un tercero, debe obtener en primer lugar su descripción, en formato WSDL. Esta descripción la obtiene a través de un servicio de descubrimiento de servicios Web, como el estándar UDDI. Otra opción podría ser obtenerla directamente del propio proveedor del servicio, en caso de que tenga su WSDL abierto públicamente (en su página, o como parte de su API). En ese caso no sería necesario el descubrimiento de forma teórica mediante el UDDI. De hecho, el perfil básico de WS-I presupone que ya tenemos acceso a la descripción WSDL de servicios del servidor.

A partir de aquí obtenemos la interfaz a todas las operaciones invocables desde el cliente, así como sus parámetros y tipos de datos. También se menciona el *endpoint*, dirección final del servidor a la cual será necesario realizar la petición.

Haciendo uso de esta información codificamos mediante XML el mensaje junto con sus parámetros y la enviamos al servidor mediante el protocolo SOAP. Este mensaje se transmite mediante HTTP (u otro protocolo como SMTP o FTP, aunque es menos usual).

El servidor acepta la petición; y escribe y envía la respuesta de forma similar: codificación XML encapsulada en SOAP y transmitida por HTTP. Con esto el cliente obtiene la respuesta a su llamada al servicio Web y se da por terminada la comunicación.

2.2.2. Transporte

La capa de transporte de la pila de los servicios web, de forma similar al estándar OSI o al modelo TCP/IP, es la encargada de asegurar el envío de la información entre los extremos finales, y al control de que esa información llega de forma correcta. No obstante, esta capa de transporte

no es sustitutiva de la capa de transporte del modelo TCP/IP, sino que hace uso de ella para transmitir la información.

Para la capa de transporte se puede utilizar cualquier protocolo estándar típicamente usado en la implementación de servidores en Internet. Los protocolos más usados para esta tarea son HTTP [7] (Hypertext Transfer Protocol, para la transferencia de páginas webs junto con sus recursos desde un servidor web a un navegador web cliente), SMTP [8] (Simple Mail Transfer Protocol, para el envío y recepción de correos electrónicos) y FTP [9] (File Transfer Protocol, para el envío y recepción de ficheros entre distintas computadoras), aunque en la práctica se podría implementar con cualquier protocolo de los usados para el envío de información por Internet.

Normalmente se utiliza HTTP sobre los demás, ya que es el que menos problemas suele dar en cuestiones de seguridad. Es extremadamente raro que se encuentren bloqueados los puertos o protocolo que utilizan (TCP en el puerto 80) en ningún firewall, y todos los ordenadores están preparados para las conexiones web a páginas en Internet, con lo cual tenemos asegurado que la conexión a los servicios web será realizada sin problemas, lo cual nos garantiza una interoperabilidad mayor que con el resto de sistemas.

Pasaré a describir con más profundidad el protocolo HTTP, y cómo es usado por los servicios web.

La versión actualmente usada es HTTP/1.1, descrita en el RFC 2616 en el año 1999. Actualmente se está estudiando por la IETF un borrador para HTTP 2.0 [10], un protocolo que sustituiría a HTTP/1.1 y que incluiría una serie de mejoras, especialmente en lo que respecta a la velocidad de las respuestas y la adaptación de la tecnología a las necesidades de las aplicaciones web más complejas, como podría ser la implementación de un *server push* o la multiplexación de las conexiones a través de los Web Sockets. Esto simplificaría la implementación de dichas aplicaciones y su velocidad, además de reducir el esfuerzo necesario para escribir una interfaz de aplicaciones basada en servicios web.

En el protocolo HTTP se definen una serie de peticiones diferentes que se le pueden realizar al servidor. Éstas se envían en unas cabeceras HTTP con el siguiente formato:

```
[OPERACIÓN] [RUTA] [VERSIÓN]
Host: [SERVIDOR]
User-Agent: [IDENTIFICADOR DEL CLIENTE]
```

donde

- [OPERACIÓN] es la operación a realizar de una lista que veremos más adelante.
- [RUTA] es la ruta relativa del recurso solicitado

- [VERSIÓN] es la versión del protocolo utilizada (normalmente HTTP/1.1)
- [SERVIDOR] es la dirección del Host servidor.
- [IDENTIFICADOR DEL CLIENTE] es una cadena de texto que identifica al cliente, ofrecida normalmente por el navegador web.

Un ejemplo de petición HTTP sería:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
```

Posteriormente se puede enviar un cuerpo del mensaje opcional, con la información necesaria para completar la petición (algunas operaciones no necesitan de dicha información adicional).

Esta petición es respondida por el servidor con una respuesta codificada de la siguiente manera:

```
[VERSIÓN] [STATUS]
Date: [FECHA]
Content-Type: [TIPO]
Content-Length: [BYTES]
[CONTENIDO]
```

donde

- [VERSIÓN] es la versión del protocolo (HTTP/1.1).
- [STATUS] es el código de estado (los más usuales son 200: OK, 404: Not found).
- [FECHA] es la fecha en formato GMT.
- [TIPO] es el tipo del contenido, en formato MIME.
- [BYTES] es el tamaño en bytes del contenido.
- [CONTENIDO] es el contenido completo a responder, en el formato especificado.

Veamos ahora un ejemplo de cómo sería una respuesta HTTP:

```
HTTP/1.1 200 OK
Date: Sun, 04 May 2014 20:00:00 GMT
Content-Type: text/html
Content-Length: 1234

<html>
<body>
<h1>Bienvenido a la página de ejemplo</h1>
</body>
</html>
```

Las conexiones bajo HTTP tienen la característica de ser *stateless*, es decir, no guardan el estado entre operaciones. Por tanto, todas las peticiones dependen únicamente de los parámetros que se pasan en dichas operaciones. Esto es algo importante para la implementación de las llamadas a servicios web, ya que de otra manera las respuestas serían inconsistentes.

De todas las posibles operaciones que se pueden realizar a través de HTTP, me voy a fijar en las cuatro operaciones que son utilizadas por los protocolos actuales para la invocación de servicios web (en concreto SOAP y REST). Estas operaciones son:

- GET: Obtener el recurso almacenado en la dirección especificada. Esta operación no realiza efectos secundarios.
- POST: Enviar información al servidor. Esto se realiza normalmente para enviar datos a través de un formulario web.
- PUT: Almacenamos la información enviada en una dirección del servidor. Normalmente no se permite y se sustituye por una conexión FTP.
- DELETE: Borra el recurso almacenado en la dirección especificada. Normalmente se realiza mediante una conexión FTP.

La semántica especificada se refiere a las operaciones HTTP tal cual se usan dentro del protocolo para la comunicación con páginas web. En el caso de los servicios web se utilizarán de forma diferente, y en el caso de REST (que veremos posteriormente), será necesario que estén disponibles todas las operaciones HTTP definidas anteriormente (no así con SOAP, que sólo utiliza peticiones GET).

Como último comentario, es importante señalar que los servicios web pueden soportarse tanto en HTTP como en su contraparte seguro HTTPS, puesto que éste sólo añade una capa de cifrado por encima de las peticiones, las cuales al descifrarse son iguales a las del protocolo HTTP [11]. Esto hace que podamos implementar servicios que utilicen datos sensibles del usuario que deban ser tratados con seguridad utilizando HTTPS y servicios web, sin necesidad de utilizar una tecnología diferente.

2.2.3. Invocación

Los protocolos de la capa de invocación de servicios web consisten en la codificación de las llamadas a las operaciones junto con sus parámetros. Estas llamadas, como ya hemos dicho, deben de poder codificarse en texto plano. Para ello, se debe serializar la información hacia texto utilizando cualquiera de los múltiples estándares y formatos diseñados al respecto. La forma más usual de hacerlo en la práctica es a través de XML o JSON.

XML (eXtensible Markup Language) es un lenguaje de marcas que se ha desarrollado por el W3C para el envío y almacenamiento de datos de forma textual [12]. El lenguaje está diseñado para que sea legible tanto por máquina como por humanos, por ello es bastante sencillo de editar tanto de forma manual como automática.

El lenguaje XML parte del desarrollo de un lenguaje generalizado de marcas por parte de la organización de estandarización ISO llamado SGML [13]. A partir de aquí surge XML, que es un rediseño del primero con muy pocos cambios que finalmente fue publicado en 1998.

XML ha ido obteniendo una popularidad creciente en los últimos años, hasta convertirse en uno de los formatos más usados para representar datos. Esto se entiende por su similitud a HTML (de hecho hay una versión de HTML llamada XHTML que es la adaptación de HTML para que sea un subconjunto de XML, evitando sus ambigüedades [14]) y por el soporte masivo del que ha disfrutado de parte de la comunidad, con gran cantidad de herramientas y desarrollos disponibles para su uso en muchos lenguajes, frameworks y plataformas.

El éxito de XML es tal que se ha convertido en la base para desarrollar gran cantidad de formatos para el almacenamiento estructurado de datos. Algunos de ellos serían:

- XHTML
- RSS (Suscripción a noticias)
- SOAP (Invocación de servicios web)
- Office Open XML (.docx, .xlsx, etc)
- OpenDocument (.odf)
- XMPP (Mensajería instantánea)
- SVG (Gráficos vectoriales)
- MathML (Extensión matemática para páginas web)
- GML (Información geográfica)
- KML (Información geográfica)

Como lenguaje de marcas que es, se basa en el uso de etiquetas para estructurar la información. Estas etiquetas se simbolizan mediante la sintaxis *<etiqueta>Contenido</etiqueta>* en el caso de las abiertas (incluyen contenido) o *<etiqueta />* en el caso de las cerradas (no incluyen contenido).

Además, cada etiqueta puede incluir atributos, con lo cual podemos resumir más información sin tener que añadir más etiquetas, simplificando la

edición y reduciendo el tamaño final. La sintaxis sería por ejemplo: <etiqueta param="42" />

Veamos ahora un ejemplo de XML más complejo:

```
<Incidencias>
    <Incidencia x=”1.15” y=”2.25”>
            <Usuario nombre=”prueba1” />
    </Incidencia>
    <Incidencia x=”43.56 y=”-3.25”>
            <Usuario nombre=”prueba2” />
    </Incidencia>
</Incidencias>
```

El tipo de etiquetas y la estructura que siguen puede ser libre o estar fija por un patrón predeterminado más estricto. Este patrón se suele llamar esquema (*XML Schema*) y lo más usual es definirlo, ya que la estructura a usar vendrá dada por los tipos de datos que tengamos que enviar para la ejecución del servicio web. La ventaja de utilizar un esquema, es que permite la validación automática del documento, lo cual ayuda mucho a la hora de la creación e implementación de documentos XML y en concreto de los servicios web [15].

La mayor ventaja de XML es finalmente la facilidad de uso y su completa adaptabilidad al tipo de información que vayamos a guardar sin ofrecer problemas.

La gran desventaja que podríamos achacar al uso de XML es que es especialmente largo comparado a formatos de almacenado de información más ligeros como JSON [16], el cual paso a describir a continuación.

JSON (JavaScript Object Notation) es un lenguaje ligero para la serialización de datos que utiliza la misma sintaxis que el lenguaje de programación JavaScript para la descripción de sus tipos de datos y objetos. La base de esta notación es el uso de arrays, tanto tradicionales como asociativos, para representar los distintos objetos que hay que codificar [17].

Un ejemplo de datos codificados en JSON sería el siguiente:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 25,
  "address": {
```

```
        "streetAddress": "123 Fake Street",
        "city": "New York",
        "country": "USA",
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```

En este caso podemos observar los dos tipos de arrays. Los arrays asociativos (denominados *Object* en JSON) siguen una estructura de clave-valor, y se utiliza para definir los distintos campos de un objeto (en el ejemplo tendríamos un objeto de tipo Persona o similar). Su sintaxis es la siguiente:

```
{  “clave1”:”valor1” , ”clave2”:”valor2”, … , ”claveN”:”valorN” }
```

Los arrays tradicionales (denominados simplemente *Array* en JSON) especifican una lista compuesta de 0 a n objetos de la misma clase. Un ejemplo se ve en los números de teléfono (*phoneNumbers*), en la cual se presenta una lista con 2 números de teléfono distintos. Su sintaxis es:

```
[ objeto1 , objeto2 , … , objetoN ]
```

Se pueden combinar cualquiera de estas dos estructuras, además de los datos básicos (Number, String, Boolean) dentro de un objeto. De esta manera podemos tener por ejemplo un Array de Array de Object o un Object cuyo uno de sus campos sea un Array que contiene Object, o cualquier otra combinación que sea necesaria para estructurar nuestra información.

JSON deriva directamente de JavaScript, aunque existen librerías para la mayoría de plataformas existentes utilizadas para la creación y consumo de servicios web [17]. No obstante, es con su uso en JavaScript donde JSON

obtiene su mayor rendimiento, ya que al usar su misma sintaxis y ser JavaScript un lenguaje interpretado, sólo es necesario una llamada a la función *eval()* ya incorporada en el lenguaje y tendremos interpretado el texto JSON como sus datos en memoria [18].

Aun así, hay que tener especial cuidado si hacemos uso de *eval()* y no de un analizador específico para JSON, ya que si los datos vienen de un servicio web no fiable puede ser una peligrosa fuente de problemas de seguridad, pues podrían inyectarnos código malicioso que se ejecutaría en nuestra máquina [18]. Por tanto sólo se debería usar cuando tenemos plena confianza en que el proveedor del servicio web enviará JSON correcto. En caso contrario sería mejor hacer uso de una librería de análisis para parsear el JSON, las cuales suelen ser especialmente eficientes en JavaScript.

En el resto de lenguajes de programación forzosamente tendremos que hacer uso de alguna librería de transformación, normalmente menos eficientes que las utilizadas para la transformación de XML. Por tanto podemos decir que JSON es el lenguaje ideal cuando el cliente que consumirá el servicio web se implemente mediante tecnología JavaScript (comúnmente será una página web que hace uso de algún servicio de terceros o que se comunica con el back-end mediante servicios web).

Para terminar haciendo una comparativa entre JSON y XML el primer punto sería el tamaño: en la mayoría de casos JSON termina ocupando un espacio menor que XML, con lo cual se consumirá un ancho de banda mucho menor si ese JSON se debe enviar muchas veces a los clientes. Esta diferencia se reduce y muchas veces incluso se elimina cuando se utiliza compresión de texto, proporcionada por algoritmos estándar de compresión de texto. No obstante esto implica un mayor tiempo de procesamiento, con lo cual JSON sigue siendo mejor en términos de consumo de espacio y ancho de banda que XML.

Con respecto a la velocidad de procesamiento actualmente JSON ofrece mejores tiempos que XML, especialmente cuando se usa la evaluación nativa de JavaScript. No obstante, la capacidad expresiva de XML es mayor, al permitir una mayor extensibilidad en la estructura y jerarquía del documento, lo cual hace que en los casos en los que la estructura es compleja ese tiempo extra de procesamiento compense.

Finalmente, los que se pueden considerar como los mejores puntos de XML con respecto a JSON son su estandarización y su legibilidad. XML está mucho más extendido que JSON, y por tanto dispone de mayor soporte tanto por organizaciones y empresas como por la comunidad. Es una tecnología más madura y más soportada, y por tanto más fácil y eficiente trabajar en ella a día de hoy. Con respecto a la legibilidad, aunque hay bastante discrepancia en este tema, generalmente se considera que XML es más sencillo de leer por un humano, y por tanto también más sencillo de editar manualmente.

Como recomendación, considero que JSON es la mejor opción para los casos en los que se realiza desarrollo web (ya que se aprovecha el hecho de ser compatible con JavaScript de forma nativa) y para cuando los datos a enviar y recibir se pueden estructurar de forma sencilla en un objeto JSON. Para los casos en que la estructura de los datos es compleja, XML tiende a ser una alternativa bastante mejor que JSON.

Una vez comentados los dos formatos más usados para la serialización de la información a enviar vamos a ver los protocolos que se usan para el envío de dicha información. Los más importantes son SOAP y REST.

SOAP (Simple Object Access Protocol) [19] es un protocolo basado en XML utilizado para enviar la información de un servicio web entre sus dos extremos (generalmente denominados *endpoints*): el emisor y el receptor. Esta transmisión es independiente del protocolo de transporte utilizado, aunque lo más común es la utilización de HTTP (o HTTPS), encapsulado dentro de un mensaje POST.

Se utiliza un tipo de mensaje con un formato estándar, que será utilizado tanto por el cliente (SOAP Request) como por el servidor (SOAP Response) para enviarse los datos. El cliente codificará en la petición la operación a invocar y sus parámetros. El servidor responderá con el resultado de la operación, codificado de manera similar.

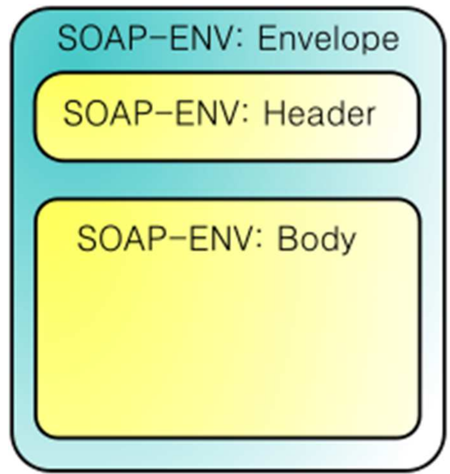


Ilustración 3 – Estructura de un mensaje SOAP

El formato fundamental de un mensaje SOAP se resume en la imagen mostrada arriba, siguiendo además el mismo esquema básico tanto para las peticiones como para las respuestas.

Se define un *SOAP Envelope*, un elemento que se utiliza para encapsular el mensaje SOAP completo. Este *envelope* se compone de una cabecera (*SOAP Header)* opcional y de un cuerpo (*SOAP Body*) en el cual se codifica la información en sí que se va a enviar. Se puede añadir también, de forma opcional, una sección de errores denominada *SOAP Fault.*

Veamos ahora un ejemplo completo del XML generado que contiene un mensaje SOAP simple, tanto de petición como de respuesta.

Soap Request:

```
POST /InStock HTTP/1.1

Host: www.example.org

Content-Type: application/soap+xml; charset=utf-8

Content-Length: 1234


<?xml version="1.0"?>

<soap:Envelope

xmlns:soap="http://www.w3.org/2001/12/soap-envelope"

soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

    <soap:Body xmlns:m="http://www.example.org/stock">

            <m:GetStockPrice>

                    <m:StockName>IBM</m:StockName>

            </m:GetStockPrice>

    </soap:Body>

</soap:Envelope>
```

Soap Response:

```
HTTP/1.1 200 OK

Content-Type: application/soap+xml; charset=utf-8

Content-Length: 1234


<?xml version="1.0"?>

<soap:Envelope

xmlns:soap="http://www.w3.org/2001/12/soap-envelope"

soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

    <soap:Body xmlns:m="http://www.example.org/stock">

            <m:GetStockPriceResponse>

                    <m:Price>34.5</m:Price>
```

```
            </m:GetStockPriceResponse>
    </soap:Body>
</soap:Envelope>
```

Al observar los mensajes se observa que la primera parte es en realidad el mensaje HTTP que se utiliza para el transporte del mensaje SOAP, que va encapsulado en él como parte de su *payload*. La petición se realiza a través de un mensaje POST y la respuesta responde directamente a ese mismo mensaje, con el XML de SOAP también dentro del payload.

En el mensaje propiamente de SOAP se observa la definición de la versión de XML y de la estructura SOAP básica (envelope y body). Dentro del cuerpo se articula el mensaje de petición/respuesta, que deberá conformar la descripción del servicio dada en el WSDL. Dentro de este cuerpo se especifican los parámetros, también en XML.

Concretamente en el ejemplo se observa que se hace una petición al precio de las acciones de IBM, y en la respuesta se devuelve dicho precio.

Voy a pasar ahora a describir en más detalle las tres partes que componen un *envelope* de un mensaje SOAP:

En primer lugar tenemos las cabeceras. Las cabeceras en SOAP son completamente opcionales, y muchos servicios web no hacen uso de ellas. No obstante hay muchas ocasiones en las que pueden ser bastante útiles, ya que sirven para especificar parámetros dependientes del sistema global, y no de la llamada concreta.

Algunos ejemplos de uso podrían ser información sobre transacciones (por ejemplo, flujos de aplicación que requieran ser transaccionales), información del estado (recordemos que SOAP y en general los servicios web conforman sistemas *stateless*), determinación de subsistemas (por ejemplo para sistemas grandes o para balanceo de carga), información de usuario (por ejemplo autentificación o localización mediante IP, entre otros) o información de pago (podríamos estar haciendo uso de una API de servicios web que precisa de pagos o cuotas para su utilización), entre muchos otros.

Estos parámetros se especificarían cada uno en una cabecera concreta que puede tener su propio formato y podemos definir de forma libre, no están predeterminados en principio por la especificación de SOAP. Por tanto podemos crear las cabeceras que necesitemos y que se adapten a la arquitectura SOA que queramos crear para nuestro sistema basado en servicios web [20].

No obstante, aunque podamos especificar las etiquetas de cabecera para adaptarlas a nuestras necesidades, dichas cabeceras podrán disponer de

unos atributos opcionales, los cuales sí están definidos de antemano por SOAP. Estos atributos pueden utilizarse para ayudar a la comunicación entre el cliente y el servidor. Los atributos son:

- soap:mustUnderstand → Se define como un valor binario 0|1 y especifica si el servidor debe entender esta cabecera de forma obligatoria o, sin embargo, puede darse que el servidor no entienda y/o ignore está cabecera.
- soap:actor → Especifica una URL a la que se enviará esta cabecera SOAP para ser procesada. Cada cabecera puede enviarse a un endpoint diferente y además puede ser diferente del endpoint destinatario del cuerpo del Web Service. El uso que se le suele dar es para poder tener un servicio de procesamiento individual para cabeceras concretas. Por ejemplo, es habitual que el servidor de autentificación de usuarios o el servidor de pagos sean independientes del servidor principal que procesa las llamadas principales de los servicios web. En caso de no especificarse el atributo esta cabecera se enviará al endpoint principal, junto con el cuerpo del mensaje SOAP.
- soap:encodingStyle → Especifica el tipo de codificación utilizada dentro del contenido encerrado en la etiqueta de esta cabecera. Este atributo no está limitado a su uso dentro de cabeceras SOAP y puede utilizarse en cualquier elemento del mensaje SOAP, como en elementos del cuerpo del mensaje. De todas maneras, como la codificación estándar de SOAP es lo suficientemente flexible, es raro cambiar el tipo de codificación de los elementos, salvo en casos muy concretos.

El componente principal de los mensajes SOAP es el cuerpo o *SOAP:Body*. El cuerpo de un mensaje SOAP (SOAP:Body) contiene la petición en sí que se realiza al servidor [21]. Debe contener un objeto, definido en el *schema* y en el WSDL, con una llamada a la operación y sus atributos. En el caso de la respuesta se opera de forma similar, con la diferencia de que este elemento del cuerpo definirá el objeto de respuesta. Los elementos del cuerpo se codifican siguiendo una codificación XML definida por el *schema* de SOAP (aunque se puede cambiar por otro diferente si así se especifica en el soap:encodingStyle). Se define normalmente un objeto de petición/respuesta (en el ejemplo GetStockPrice), que puede a su vez contener otros elementos (no sería obligatorio). Estos elementos pueden ser simples (tipos finales escalares tales como enteros, dobles, cadenas o enumerados, entre otros) o compuestos (tipos estructurados o arrays, por ejemplo) y sirven de uso como parámetros de entrada de la operación de petición o como los propios datos de respuesta tras la ejecución de la misma. Para una completa descripción del esquema de codificación se puede observar la definición del *schema* de SOAP [22] [23].

Por último, se pueden especificar errores usando el elemento *SOAP:Fault* [24]. Este elemento es opcional, y sólo se usara cuando queramos definir un

error ocurrido durante la ejecución del servicio web. El elemento fault tiene 4 atributos XML a definir. Estos son:

- **faultcode**: Código único que identifica al error. Existen varios códigos de error reservados ya definidos, aunque se puede ampliar y añadir nuevos códigos según necesitemos. Los códigos predefinidos son:
    - **VersionMismatch**: No se entiende el mensaje SOAP (SOAP Envelope). Generalmente por una diferencia de versiones (SOAP 1.1 frente a SOAP 1.2 por ejemplo) o se ha especificado mal el espacio de nombres de SOAP.
    - **MustUnderstand**: El servidor no tiene definida una de las cabeceras enviadas por el cliente con el atributo mustUnderstand=1.
    - **Client**: Error por parte del cliente. Generalmente se da cuando el mensaje SOAP está mal formado o no se ajusta al *schema* alguno de los elementos.
    - **Server**: Error que se da cuando el mensaje es correcto pero el servidor ha tenido algún problema y no ha podido ejecutarse. Normalmente se da cuando el error viene por parte del procesamiento de SOAP o es desconocido. Si es un error específico de la aplicación se debería usar un código propio que identifique a dicho error.
- **faultstring**: Cadena que especifica el error de forma entendible por humanos.
- **faultactor**: URL del servidor o endpoint que ha causado el error. Es opcional y se utiliza normalmente cuando el causante del error no es el destinatario final del mensaje.
- **detail**: Es opcional y se utiliza para dar información del error concreta, específica de la aplicación. Para ello se pueden utilizar elementos definidos por el usuario (de forma similar a las cabeceras) denominadas entradas (*entries*) o especificar el detalle directamente de forma textual.

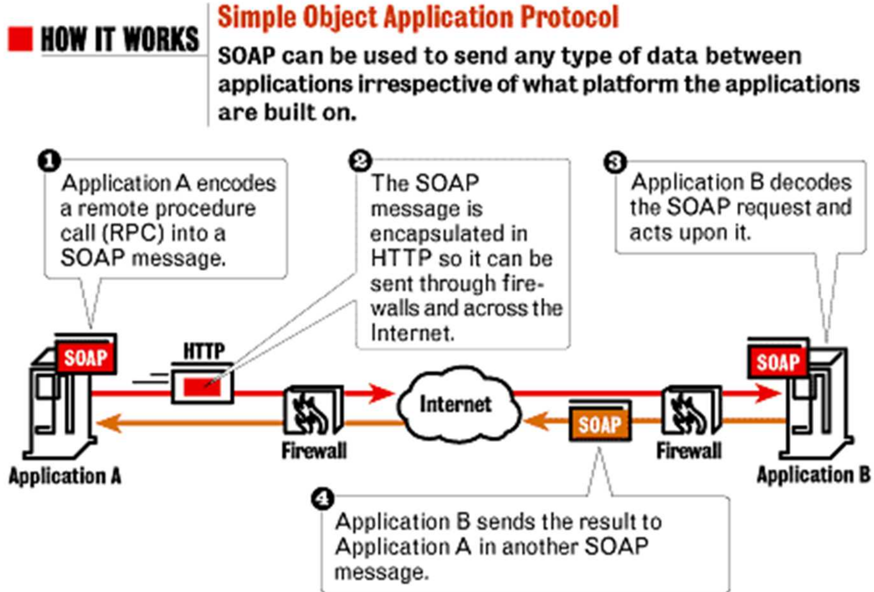


Ilustración 4 – Comunicaciones en SOAP

Después de haber visto cómo funciona el protocolo SOAP a grandes rasgos, veamos el funcionamiento de su principal competidor: REST.

REST no es estrictamente hablando un protocolo, sino más bien un esquema de arquitectura software para la implementación de servicios web [25]. Este esquema se utiliza para la invocación de los servicios web del servidor a través de una semántica propia, basada principalmente en el uso de URIs para acceder a recursos mediante el protocolo HTTP. REST se utiliza en multitud de APIs y sistemas en Internet. Por poner un ejemplo, es la base de muchas APIs de importantes páginas web como Facebook o Twitter.

El termino REST significa "Representational State Transfer", o Transferencia de estados representacional y fue descrito en el año 2000 por el Dr. Roy Fedling, uno de los principales miembros de la especificación HTTP. Su nombre se basa en la filosofía básica de REST: en una URI base disponemos de una serie de recursos (ej: www.ejemplo.com/usuarios) los cuales pueden ser accedidos por el cliente. Estos recursos son una representación de la información accedida (normalmente en algún formato estándar, como veremos posteriormente). Estos recursos, con su información actual, representan estados concretos del sistema (ej: los datos de los usuarios en la URI anterior). Estos recursos pueden ser modificados (inserciones o eliminaciones, por ejemplo) mediante llamadas a esos recursos, lo cual supondrá una modificación o transferencia del estado del sistema (ej: un usuario cambia alguno de sus datos, lo que hará que cambie su representación). Todo esto, al juntarse, es lo que da lugar al nombre del acrónimo de las tecnologías: REST.

Aunque REST se puede utilizar como framework base para la implementación de muchos sistemas basados en web (en concreto HTTP), el uso principal y más provechoso es la implementación de servicios web. Lo

que REST propone como modelo de uso para los servicios web es organizar el sistema en una serie de recursos HTTP accesibles mediante su URI, a los cuales se le pueden aplicar las cuatro principales operaciones de acceso a datos (consulta, inserción, modificación y eliminación), generalmente denominadas operaciones CRUD y soportadas normalmente por un sistema gestor de bases de datos.

Para ello hacemos uso de los verbos u operaciones HTTP, aplicados a cada uno de los recursos del sistema. Hay dos tipos principales de recursos: las colecciones y los elementos individuales. Esto se ve claramente con un ejemplo: una base de datos de usuarios consultable mediante REST.

La colección es una lista podría ser accedida mediante la URI www.ejemplo.com/usuarios y contendría las operaciones básicas de la lista completa de usuarios, siendo el listado y consulta las más habituales. Luego podríamos querer acceder a un usuario concreto, bien sea para modificarlo, añadirlo o eliminarlo. En ese caso accederíamos a su URI particular, algo similar a www.ejemplo.com/usuarios/1234, donde 1234 sería el identificador único de dicho usuario. Para cada una de las operaciones CRUD sobre colecciones o elementos individuales accederíamos a dicho recurso mediante una operación HTTP diferente. En la siguiente tabla se pueden ver las distintas operaciones aplicadas a colecciones o elementos:

	GET (Consulta)	**PUT (Modificación)**	**POST (Inserción)**	**DELETE (Eliminación)**
Colecciones	Consulta la representación de los elementos de la colección (puede parametrizarse o listarse completa)	Modifica la colección por otra diferente, introducida por el usuario en la petición.	Inserta un nuevo elemento en la colección, especificada por el usuario en la petición.	Borramos la colección completa (o una subcolección si parametrizamos)
Elementos	Se devuelve la representación de este elemento particular.	Modificamos (o creamos si no existe) el elemento especificado en la petición	Modificamos el elemento para convertirlo en colección y a su vez añadimos el elemento especificado en la petición.	Borramos el elemento especificado de la colección.

Tabla 1 – Operaciones REST

Como es comprensible, muchas veces algunas de estas operaciones no se implementarán para algunos de los elementos o colecciones, según el

diseño que queramos que tenga nuestra API, bien porque algunas operaciones no tienen sentido (el POST en elementos únicos casi nunca se usa, o el PUT en colecciones, entre otros) o porque no deben de poder realizarse para mantener la integridad de la estructura de la información por falta de permisos (por ejemplo, un DELETE tanto de colecciones o individuales al recurso de usuarios por parte de un usuario cualquiera sería fatal para la seguridad de nuestro sistema) o por cualquier otra razón de diseño de nuestra API basada en servicios web.

La representación de la información de un recurso normalmente sigue un estándar, que deberá ser acordado entre el cliente y el servidor. Normalmente se utilizan estándares web, como los ya vistos XML y JSON, aunque en la práctica cada vez más servicios *RESTful* (nombre que se suele dar a los servicios basados en REST) utilizan la representación en JSON de los datos. Además, muchos sistemas ofrecen la oportunidad de seleccionar que tipo de representación utilizar, para mayor comodidad hacia el cliente.

Los parámetros incluidos en las peticiones se suelen añadir como parte de la URL y no deben confundirse con los identificadores de elementos particulares. Un posible ejemplo para acceder a usuarios registrados entre 2 fechas (representadas como marcas de tiempo) sería:

```
www.ejemplo.com/usuarios?since=123512538127&to= 234623649238
```

Veamos ahora un ejemplo de llamada REST, una modificación de un usuario concreto (1234), con una representación en JSON:

```
PUT /usuarios/1234 HTTP/1.1
Host: www.ejemplo.com
Content-Type: application/json; charset=utf-8
Content-Length: 5678


{
  "firstName": "John",
  "lastName": "Doe",
  "age": 25,
  "address": {
    "streetAddress": "123 Fake Street",
    "city": "New York",
    "country": "USA",
  },
```

```
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ]
}
```


La principal diferencia entre SOAP y REST es la arquitectura de la aplicación. Mientras que en REST la arquitectura se basa en el acceso a los distintos recursos mediante los verbos HTTP, en SOAP podemos implementar cualquier llamada a cualquier método arbitrario, siempre o cuando lo definamos en la API y lo añadamos a la correspondiente descripción WSDL. Esto puede dar lugar a problemas de diseño al realizar una API REST, ya que muchas operaciones que no son sobre recursos no se podrán hacer, y habrá que inventar trucos para simular estas operaciones mediante recursos ficticios.

Un ejemplo de esto podemos verlo en un sistema de compra online. En una API basada en SOAP podríamos tener un método remoto denominado *comprar()* con una serie de parámetros de compra tales como las líneas del pedido o los datos de pago del usuario. En REST no podríamos hacer esto, ya que solo podemos acceder a recursos, lo cual obligará a complicar el diseño de la API, en este caso concreto podríamos crear un nuevo recurso denominado *ordenCompra*, y encapsular una compra como una llamada REST a la operación POST (inserción) de la colección de recursos de ordenCompra.

Con respecto a cuál de los dos sistemas para la invocación de servicios web es mejor, no hay una respuesta clara. REST es un sistema más simple y más fácil de crear y mantener. No obstante es menos flexible y menos organizada, lo primero por no permitir métodos arbitrarios y lo segundo por no estar organizado con unos protocolos y estándares bien definidos y controlados, algo que sí ocurre en SOAP.

Por tanto, podemos resumir diciendo que SOAP es un protocolo más asentado y mejor controlado y estandarizado (aunque de hecho es posterior en el tiempo) pero más lento y más complejo de implementar y mantener. REST es todo lo contrario, una arquitectura ligera y simple, más

fácil de usar e implementar pero menos flexible y más caótico. Por tanto lo habitual es utilizar REST en aquellos sistemas de arquitectura sencilla (o que se adapte bien al sistema de recursos), y utilizar SOAP para sistemas con arquitecturas complejas, especialmente si se requieren operaciones arbitrarias.

2.2.4. Descripción

La descripción de los servicios accesibles desde el exterior mediante servicios web o cualquier otra tecnología más antigua (por ejemplo, CORBA) es indispensable para que los clientes puedan reconocer de forma simple y a poder ser automática cuales son las distintas operaciones o métodos que se pueden invocar, y de qué manera se invocan.

Por tanto, todos los sistemas que se basan en llamadas a procedimientos remotos o, de forma más general, a cualquier tipo de comunicación software realizada de forma remota, suelen utilizar algún tipo de lenguaje o protocolo de descripción de interfaces [26]. Estos lenguajes permites ofrecer una descripción exhaustiva de la interfaz que ofrece al exterior un sistema determinado y generalmente crean de forma automática un esqueleto para que el cliente pueda llamar a estos servicios de forma aparentemente local, enmascarando la llamada remota. Estos lenguajes se suelen llamar *IDL* (Interface Description Languages) y se utilizan también dentro de la pila de protocolos de Web Services para la descripción de los servicios a ser invocados como parte de las capas anteriores (SOAP o REST, en su mayor parte).

El principal lenguaje utilizado para la descripción de servicios web se denomina *WSDL* (Web Service Description Language) y consiste en un documento XML con una sintaxis determinada que permite especificar la interfaz de los distintos métodos que se pueden llamar a un servicio web. Actualmente se utiliza la versión 2.0 y es una recomendación por parte del W3C [27].

Generalmente WSDL se utiliza en combinación con SOAP, ya que es muy sencillo realizar una llamada SOAP a partir de la interfaz de un método definida en un documento WSDL. Se podría usar de hecho con otros protocolos para la invocación de servicios (como REST) o incluso con otros sistemas RPC, pero no tendría mucho sentido, ya que para estos podríamos encontrar otras herramientas de descripción mejor adaptadas a estas tecnologías. Además, una de las grandes ventajas del uso de WSDL junto con SOAP es que estas tecnologías encajan en la pila de protocolos especialmente bien, de tal manera que permiten la generación casi automática de las llamadas de creación y consumo de los servicios, a partir únicamente de la descripción WSDL.

Pasaré ahora a describir los distintos elementos (como etiquetas XML) que componen un documento WSDL [28]:

- **Description**: Es el nodo raíz del documento WSDL (recordemos que es

un XML) y todos los demás elementos se insertarán dentro de éste. Este elemento puede incluir varios atributos. Los más importantes son *name*, que especifica el nombre del Web Service (debe ser único en el sistema) y *targetNamespace*, que especifica cual es la ruta del archivo dentro del espacio de nombres del sistema. Además, se pueden especificar algunos espacios de nombres de utilidad para el documento. Los más utilizados son *xlmns:soap* y *xmlns:xsd* para poder especificar SOAP y XML Schema respectivamente.

- **Types**: Se especifican los tipos de datos que se utilizarán como entradas o salidas de las funciones definidas en la interfaz WSDL. Estos datos se describen utilizando XML Schema (de forma general y recomendada por la W3C, una implementación propia podría utilizar otro formato). Se puede hacer la especificación directamente en el documento WSDL o añadir una referencia a un documento XSD con la especificación en formato XML Schema. También se pueden combinar ambos enfoques. Para especificar un XSD externo, se utilizará la etiqueta *schema* y en el atributo *targetNamespace* la ruta al documento XSD que importaremos dentro del esquema de tipos de nuestro documento WSDL. Para crear nuevos tipos en el WSDL usaremos la etiqueta *element* y la especificaremos usando directamente XML Schema.
- **Message**: Se utiliza para especificar los mensajes a enviar, tanto de petición como de respuesta. Cada una de las operaciones se compondrá por tanto de un par de mensajes unidos entre sí. Por convención, estos mensajes se suelen llamar *NombreOperacionRequest* y *NombreOperacionResponse*. Cada uno de estos mensajes contendrá una lista de sus atributos mediante etiquetas *part*. En estas etiquetas aparece el nombre del atributo (entrada en el caso de peticiones y salida en el caso de respuestas) y el tipo del atributo, sacado de la lista de tipos definida o directamente de los tipos básicos en XSD.
- **PortType**: También puede especificarse como *interface*, según la versión de WSDL utilizada. Sirve para especificar la interfaz final de una operación, como un conjunto de mensajes entrelazados. Aunque se pueden realizar algunos esquemas diferentes, lo más habitual y más útil es utilizar el esquema de *request-response*, por el cual el servidor recibe el mensaje de petición (request) y lo responde con su correspondiente respuesta (response). Para especificar la interfaz de estas operaciones se utilizan las etiquetas *input* y *output*. En estas etiquetas se especifica el mensaje que se usa de entrada y de salida, como referencia a un mensaje creado en la sección de mensajes anterior dentro del documento WSDL. Las otras posibilidades son *one-way* (petición sin respuesta, no podemos asegurar que se haga correctamente y sólo sirve para cuando no necesitamos devolver datos), *notification* (el propio servidor envía una respuesta sin necesidad de una petición por parte del cliente, puede ser útil en algunos casos para hacer push desde el servidor pero no encaja en la arquitectura típica de los servicios web) y *solicit-response* (se cambian los papeles y el servidor manda una

respuesta o notificación, esperando además una entrada posterior del cliente).

- **Binding**: Sirve para unir la definición de la interfaz de una operación con su método final de invocación, que generalmente será SOAP sobre HTTP, aunque se pueden utilizar algunos otros. Se pueden generar además múltiples *bindings* sobre la misma operación, de tal manera que pueda decidirse por el cliente varias maneras se invocarla (SOAP o HTTP puro, por ejemplo). Por defecto el estándar WSDL sólo incluye como posibilidades distintos *binding* para SOAP, aunque se pueden añadir más con personalizaciones.
- **Service**: Es el último conjunto de etiquetas, que se utiliza para unir una operación con un *binding*, y dotarle por tanto de entidad como servicio completo. En la descripción del servicio se añade además el endpoint, o dirección final a la cual se debe conectar el cliente para acceder al servicio y, de forma opcional aunque recomendable, una descripción en lenguaje natural de la funcionalidad y uso del servicio (para ser leído por humanos).

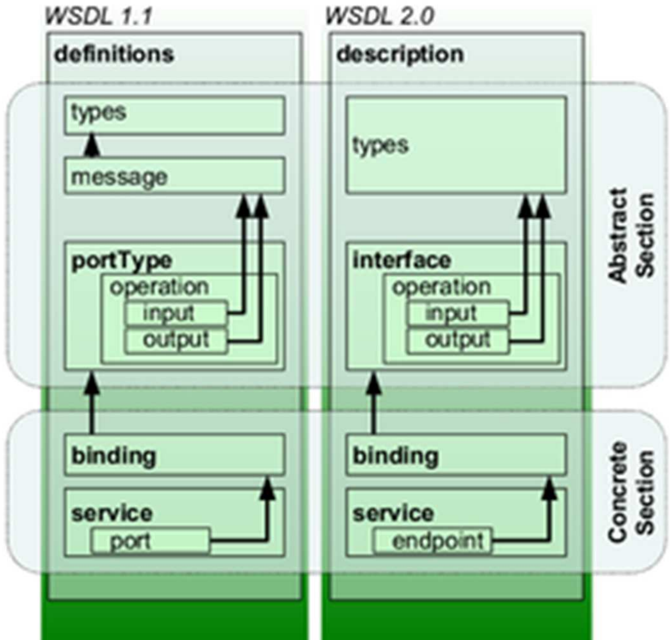


Ilustración 5 – Estructura de un mensaje WSDL

Veamos ahora un ejemplo completo de un documento WSDL que recoge todos estos elementos:

```
<description name="HelloService"
  targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<message name="SayHelloRequest">
  <part name="firstName" type="xsd:string"/>
</message>
<message name="SayHelloResponse">
  <part name="greeting" type="xsd:string"/>
</message>

<portType name="Hello_PortType">
  <operation name="sayHello">
    <input message="tns:SayHelloRequest"/>
    <output message="tns:SayHelloResponse"/>
  </operation>
</portType>

<binding name="Hello_Binding" type="tns:Hello_PortType">
<soap:binding style="rpc"
  transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="sayHello">
  <soap:operation soapAction="sayHello"/>
  <input>
    <soap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:examples:helloservice"
      use="encoded"/>
  </input>
  <output>
    <soap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:examples:helloservice"
```

```
      use="encoded"/>
    </output>
  </operation>
  </binding>

  <service name="Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
    <port binding="tns:Hello_Binding" name="Hello_Port">
      <soap:address
        location="http://www.examples.com/SayHello/">
    </port>
  </service>
</description>
```

Comentando el ejemplo, lo primero que vemos es el apartado de *description* como nodo raiz, en el cual importamos los distintos namespaces a usar y damos nombre al WSDL (HelloService). No se define una sección de tipos, ya que sólo se van a usar strings básicos de XML Schema (xsd:string).

Posteriormente se define un par de mensajes request-response: SayHelloRequest y SayHelloResponse. Para la petición se envía como parámetro *firstName* y la respuesta incluye una cadena con el saludo *greeting.*

Después se define la operación *sayHello* mediante la etiqueta *portType*, que une el mensaje de petición al de respuesta. Tras esto se define un *binding* SOAP sobre HTTP (esto se especifica en *transport*) llamado *Hello_Binding* para la operación *sayHello*.

Por último definimos el servicio *Hello_Service* en el cual especificamos el endpoint www.examples.com/SayHello para la operación *sayHello* con el único *binding* del que dispone, *Hello_Binding*, que se invoca mediante SOAP sobre HTTP.

Con respecto a las tecnologías REST no hay un lenguaje de descripción tan claro como WSDL. La mayoría de las veces las APIs implementadas con REST se consideran autodescriptivas, ya que la semántica de las operaciones viene dada por la operación invocada (listar, insertar, modificar o eliminar) y el nombre del recurso accedido. Esto hace que muchos programadores consideren que no es necesario usar un lenguaje

de descripción de servicios cuando éstos se implementan según la filosofía REST.

No obstante, aunque sí que es cierto que el poder descriptivo de un servicio REST es mayor de por sí al de un servicio clásico mediante RPC, sería imposible usar realmente estos servicios sin una descripción de su uso. Por ello se utilizan principalmente 2 enfoques, que pueden usarse por separado o de forma conjunta.

El primer enfoque y el que suele considerarse preferido por la comunidad REST es la realización de una documentación de la interfaz del sistema, a modo de referencia de la API para los programadores que llamen a dichos servicios. Esto se suele realizar de forma automática a partir de comentarios escritos a mano en el propio código, siguiendo el ejemplo de sistemas como JavaDoc en Java o los comentarios XML en .NET. El problema de este sistema es que no genera documentos tratables de forma automática por sistemas software como frameworks o middlewares. Sin embargo, y como es usual ofrecer una documentación de este estilo en cualquier sistema de cara a los programadores, no está de más añadirla, pero no se puede considerar un sistema de descripción de servicios riguroso. Algunos de estos sistemas de documentación automática son Swagger [29] y JSONDoc [30].

El otro enfoque es hacer algo similar a WSDL, y utilizar un lenguaje formal de descripción de interfaces. Aunque no se suele usar y por tanto no hay ningún formato que haya alcanzado el nivel de estándar, puede ser interesante en muchos casos contar con un lenguaje que describa de forma precisa estos servicios. Algunos de estas propuestas son: WSDL 2.0 [31], que se puede adaptar para describir servicios REST, y WADL [32].

2.2.5. Descubrimiento

La última capa dentro de la pila de protocolos de servicios web es el descubrimiento de servicios. El descubrimiento consiste en la capacidad por parte de un consumidor potencial de servicios web de encontrar servicios publicados por una organización de terceros para poder ser utilizados (consumidos) por su parte.

Esta etapa de descubrimiento se puede realizar de forma automática o de forma manual. En el caso de realizarse de forma manual no se podría considerar realmente como parte de la pila de protocolos, ya que requiere de un trabajo previo por parte del desarrollador. En este caso las opciones pueden ser varias, aunque siempre dependen de que haya descripciones del servicio, estén en el formato que estén. Algunas de estas opciones podrían ser:

- Conocer de antemano la dirección del documento (WSDL/WADL) que describe el servicio.

- Encontrar la descripción a través de un motor de búsqueda (ej: Google) o de la página web de la organización o servicio al cual queremos conectarnos (ej: Twitter).
- Realizar una búsqueda automatizada de los documentos descriptores a través de la web.
- Buscar a partir de algún directorio centralizado.

El caso de realizar el descubrimiento del servicio de forma manual es lo más usual, ya que cuando vamos a conectarnos a un servicio web normalmente lo realizamos para añadir una funcionalidad a nuestro software o cubrir alguna nueva necesidad, y por tanto podemos permitirnos descubrir de forma manual estos nuevos servicios, para encontrar el que más nos interesa.

Antes de empezar a hablar sobre cómo se puede realizar el descubrimiento de servicios de forma automática es importante hablar del registro UDDI (Universal Description, Discovery and Integration) de OASIS [33]. En este registro se incluyen una serie de descripciones WSDL para multitud de servicios web de diferentes organizaciones y unos sistemas para permitir búsquedas bajo distintos criterios. Aunque actualmente sigue operativo ya se considera obsoleto por la propia organización y por todos sus principales usuarios y sólo sirve de repositorio para proyectos descontinuados, servicios de ejemplo y cosas similares [34].

Aun así, la idea de un sistema similar a UDDI permanece, aunque para tener una utilidad real es necesario ser utilizado a un nivel más local. Por tanto se suelen usar repositorios o registros similares dentro de una empresa u organización para interconectar sus distintos sistemas utilizando arquitecturas SOA. Esto suele realizarse de forma automatizada a partir de unos middleware que se denominan ESB (Enterprise Service Bus) [35]. Los ESB ofrecen multitud de funcionalidades necesarias para interconectar distintos sistemas formando una arquitectura global de la organización en base a dichos servicios (una arquitectura SOA), siendo una de ellas el descubrimiento de los servicios a los que hemos de conectarnos de forma dinámica.

2.3. Librerías de Twitter

2.3.1. Introducción

Twitter es la base utilizada durante el proyecto para el envío y recepción de mensajes que codifican las incidencias, como método de transporte entre el cliente y el servidor. Por eso es importante comentar qué es Twitter y como se hace uso de sus servicios web desde dentro de la aplicación.

Twitter es uno de los servicios más importantes e influyentes en Internet a día de hoy [36], con una base de usuarios activos de aproximadamente 218 millones y unos 500 millones de mensajes (tweets) diarios enviados de media.

Hay muchas maneras de definir a Twitter y en mi opinión todas ellas pueden considerarse correctas, ya que es un servicio realmente novedoso y con muchas aplicaciones. La definición que se considera oficial y que más gente utiliza es la consideración de que Twitter es una red social de microblogging. Esto significa que en Twitter los usuarios se agrupan como en una red social y su principal función es la de publicar mensajes llamados tweets con un límite de 140 caracteres. Estos tweets sólo permiten enviar texto plano, aunque existen varios servicios (incluido uno oficial, pic.twitter) que permiten codificar contenidos multimedia, usualmente imágenes, dentro del tweet, aunque para ello se requiere utilizar parte de los caracteres del mensaje en dicha codificación [37].

Twitter tiene una serie de características que hacen que se una de las mejores elecciones para la implementación del proyecto, que comentaré a continuación.

La primera de ellas es su estructura asimétrica: Twitter utiliza una estructura asimétrica para las relaciones entre los usuarios. La mayoría de las redes sociales (ej: Facebook) utilizan la metáfora de “amigo” para definir una relación entre 2 usuarios. Esta relación es por tanto simétrica, si Usuario A es amigo de Usuario B, Usuario B también lo será de Usuario A. En Twitter la relación entre usuarios se define en términos de “followers” y “followings”. Un usuario puede seguir (en inglés follow) a múltiples usuarios, convirtiéndose así en un follower para dichos usuarios, y dichos usuarios se convertirán en sus following. Esto significa que este usuario escuchará todas los tweets que hagan cada uno de los usuarios a los que sigue. Está relación no es simétrica, ya que estos usuarios seguidos por él no tienen por qué seguir a su vez al usuario que les sigue, y pueden decidir seguir a usuarios diferentes.

La mayor consecuencia de esto es la aparición de cuentas de organizaciones, empresas o incluso famosos que tienen muchos seguidores pero siguen a pocos usuarios. Este esquema es uno de los pilares de Twitter y se ajusta mucho a lo que queremos para la aplicación, pues podemos tener una cuenta de Twitter que tenga muchos seguidores y que recibirá los tweets de muchos usuarios, que serán los clientes del sistema, pero que no necesita una comunicación de vuelta hacia ellos.

Posteriormente, y relacionado con lo anterior tenemos un mecanismo en Twitter que se conoce como menciones. Los usuarios en Twitter se definen como @nombreUsuario, y con dicho nombre tenemos un identificador único de usuario. Un tweet puede incluir menciones a distintos usuarios, de manera que si incluimos su nombre de usuario en el tweet recibirá el tweet independientemente de si nos sigue o no, además

podrá consultarlo explícitamente en su lista de menciones. Esto tiene varios usos en un comportamiento normal en Twitter, aunque el que más interesa para el desarrollo es la posibilidad de recibir todas las incidencias como menciones hacia la cuenta de la aplicación. De esta manera no necesitaremos monitorizar todos los tweets de nuestros usuarios (lo cual implicaría seguirlos, cosa que no queremos, además de que a la hora de recuperar las incidencias nos llevaría a una mayor carga de procesamiento de datos) y podremos obtener únicamente las menciones a nuestra cuenta.

El último punto a tener en cuenta para el uso de Twitter en el proyecto, aunque de hecho es uno de los más importantes es la abertura de su ecosistema. Twitter es una plataforma abierta al uso de su sistema por parte de terceros, ofreciendo una buena API para programadores basada en servicios web y promocionando su uso, algo que se ve claramente con el número de aplicaciones que hay que utilizan integración con Twitter, tanto en la web como en dispositivos smartphone.

La comunicación con la API de Twitter se realiza a partir de dos APIs principales, la API REST [38] y la API de Streaming [39]. Con la primera se pueden hacer peticiones REST a los distintos recursos (tweets, usuarios, etc.) como hemos visto anteriormente. Con la API de Streaming podemos suscribirnos a un stream que nos mandará la información a la que nos hayamos suscrito en tiempo real, sin tener que realizar peticiones REST periódicamente. Las dos APIs se pueden utilizar simultáneamente en la misma aplicación, usándose la API de Streaming para conexiones persistentes (por ejemplo, monitorizar los tweets entrantes de una cuenta) y la API REST para peticiones singulares (por ejemplo, obtener el nombre de usuario del emisor de un tweet).

Ambas APIs hacen uso de una fuerte seguridad, siendo obligatorio el uso de HTTPS y la autentificación a 3 vías mediante OAuth, necesaria para que una aplicación de un tercero pueda realizar operaciones en Twitter en nombre de un usuario con la suficiente seguridad y sin acceder a sus credenciales [40] [41].

2.3.2. Autorización OAuth

Dado que el funcionamiento de la autentificación OAuth es bastante complejo y muy importante en las conexiones seguras a APIs de terceros, pasaré a explicar su funcionamiento.

OAuth es un estándar en seguridad informática para permitir la autorización a una aplicación o sistema de actuar en nombre de un cliente (normalmente el usuario final) en el sistema de una tercera parte. Esto se suele utilizar en las APIs de sistemas que permiten a desarrolladores hacer aplicaciones externas que realizan ciertas funciones en nombre del usuario final. Un ejemplo muy claro lo podríamos encontrar en la

integración de APIs de redes sociales o de sistemas de almacenamiento en la nube.

OAuth se creó en 2007 para solucionar el problema descrito anteriormente, ya que no había ningún protocolo ni sistema que realizase una autorización a acceso a un sistema de terceros. La versión que se creó fue OAuth 1.0, la versión más extendida a día de hoy, ya que aunque ha aparecido recientemente la versión 2.0 estás no son compatibles entre sí y OAuth 2.0 ofrece unos esquemas de autentificación que no encajan tan bien con el uso típico que se le suele dar a OAuth [42]. Esto, unido a una falta de seguridad y una complicación excesiva del proceso de autentificación, hace que no termine de tener una aceptación mayoritaria por parte de la comunidad de desarrolladores.

Para conseguir la autentificación OAuth se pueden utilizar varios mecanismos. El más usado, y el que usaré para conectar con Twitter, es un sistema de 3 vías (denominado en inglés 3-legged) que se resume de la siguiente manera:

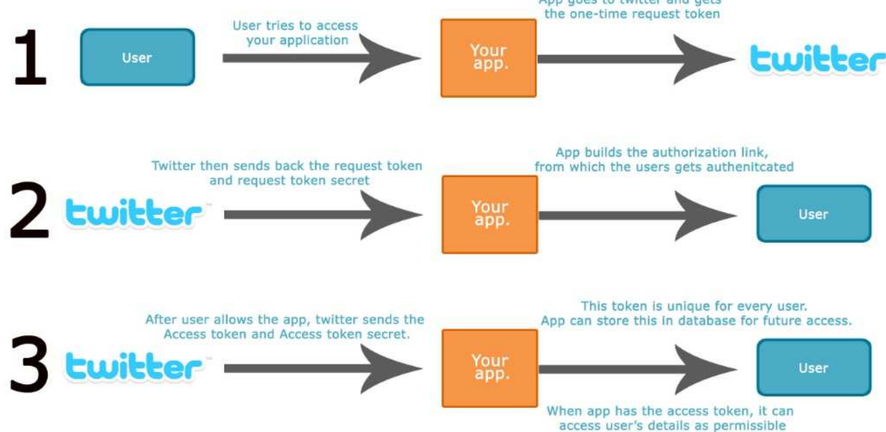


Ilustración 6 – Esquema de autentificación OAuth

Como se puede ver el cliente pide a la aplicación que se conecte en su nombre, para lo cual ésta inicia una petición OAuth con el servidor. Para ello le pide un *Request Token / Request Token Secret* de uso único. Con este Request Token se genera una URL de autentificación, que es accedida directamente por el usuario (con un navegador web). En esta página el usuario introduce sus credenciales (normalmente usuario y contraseña), que sólo serán vistos por el sistema final y no por la aplicación intermedia. Tras comprobar que son correctos los credenciales y el Request Token, el servidor da la petición por aceptada y genera un *Access Token / Access Token* Secret. Este par de claves finales pueden ser usadas por la aplicación para conectarse en nombre del usuario, y unen de forma inequívoca al usuario con la aplicación, de cara al sistema final.

Una vez obtenido el token de acceso se utilizará para firmar las peticiones realizadas a la API (usando el secreto o clave privada), ya sean a la API basada en REST o a la API de Streaming. Las respuestas de ambas API serán enviadas cifradas con la parte pública del token de acceso, de forma que sólo la aplicación con acceso a dicho token podrá descifrarlas.

Las ventajas de usar OAuth con respecto a una autentificación directa en el sistema final son varias:

- La aplicación no llega a conocer en ningún momento las credenciales del usuario en el sistema proveedor.
- Los tokens de acceso pueden ser revocados por el usuario o el sistema final, en caso de que se quiera negar el acceso a los datos del cliente a la aplicación.
- Una vez obtenido el token de acceso se considera seguro por su esquema criptográfico, por tanto se puede usar directamente sin necesidad de negociar la conexión cada vez que accedamos.
- En caso de problemas o abuso de la autentificación por parte de una aplicación es más fácil identificarla y sobre todo, más fácil de revocar dichos permisos.

2.3.3. API REST

La API REST de Twitter sigue la estructura de una API REST convencional. No obstante, en esta API sólo se permiten operaciones GET (para obtener recursos) y POST (para añadir o modificar recursos). Esto es así ya que por diseño no es interesante para Twitter permitir el borrado de información de forma automática mediante su API. No obstante, algunos recursos pueden ser borrados de forma lógica (que no física) accediendo a una operación POST con la URL *recurso_a_eliminar/destroy*.

De forma general, para realizar la llamada REST se hace una petición HTTP GET/POST con la URL del recurso a acceder y posteriormente se añaden los parámetros a enviar directamente en la URL del recurso, dentro de la lista de parámetros soportados por la operación, que se pueden consultar en la documentación oficial de la API de Twitter [43]. Un ejemplo para acceder al Timeline de un usuario sería:

```
GET
https://api.twitter.com/1.1/statuses/mentions_timeline.json?count=2&si
nce_id=14927799
```

Los distintos recursos que se pueden acceder desde la API son muy diversos y se pueden consultar directamente en la documentación de la API de Twitter [43]. Algunos de los más importantes son:

- **GET statuses/mentions_timeline**: Devuelve una lista de menciones del usuario autentificado. Cabe mencionar que dentro de la API de Twitter

los tweets son llamados status, y todas las operaciones sobre tweets tienen como URL status/algo. El resto de timelines se pueden consultar de forma similar.

Algunos de los parámetros más importantes (aunque opcionales) son:

- **count**: El número de tweets a devolver.
- **since_id**: El id (único) de tweet a partir del cual empezará la lista.
- **max_id**: El id de tweet máximo que se permite en la lista.

- **POST status/update**: Publica un nuevo tweet desde la cuenta del usuario. El único parámetro importante es status, el propio texto del tweet.
- **GET user**: Comienza un Stream de usuario. Esto hace que comience el uso de la API de Streaming, la cual veremos en el apartado siguiente.
- **GET direct_messages**: Obtenemos los mensajes directos (mensajes que no son tweets y actúan de forma similar a los mensajes privados de un foro) del usuario. Los parámetros son los mismos que en la obtención de timelines.
- **GET followers/list**: Se obtienen una lista con los followers del usuario.

El formato utilizado para codificar la información de los distintos recursos es JSON. Cada uno de los recursos tiene su propia representación, en la cual se incluyen todos los atributos de dicho recurso o entidad, los cuales se pueden consultar en la API de Twitter [44]. En el caso de las peticiones GET obtendremos como respuesta el JSON con el objeto o la lista de objetos recibidos. En el caso de las peticiones POST tendremos que enviar el JSON del objeto a añadir como parte del cuerpo del mensaje HTTP.

Una parte muy importante de esta API REST, es el hecho de que todas las operaciones tienen unas limitaciones de uso bastante estrictas [45]. Esto es así para evitar un abuso del sistema. Estas limitaciones se comprueban en una ventana de 15 minutos. Si superamos el número de peticiones de una operación determinada en ese tiempo no podremos realizarla hasta que se resetee el número de usos restantes al terminar dicha ventana de tiempo, y recibiremos un error “Too Many Requests”.

Los límites se definen por aplicación/operación/usuario. La asociación aplicación/usuario se realiza como ya se ha dicho a través del token de acceso OAuth. Por tanto Twitter comprueba que para cada token no se supere el límite en cada una de las operaciones de forma individual. Esto permite que si un usuario supera un límite no penalizar a un usuario que no lo ha sobrepasado, o que si un usuario ha sobrepasado el límite de una operación (ej: consultar tweets), pueda seguir haciendo otras operaciones (ej: enviar tweets).

Las limitaciones exactas de cada operación pueden consultarse en la documentación oficial [46], aunque entran principalmente en 2 rangos: 15 peticiones por ventana o 180 peticiones por ventana. Normalmente el rango más restrictivo se reserva para aquellas operaciones más pesadas o para las más comprometidas, dejando el rango más permisivo para las demás.

Para poder evitar el impacto de las restricciones se pueden realizar varias técnicas que liberan la cantidad de llamadas a realizar.

La primera es el cacheo de resultados. Cuando obtengamos los resultados de una llamada a la API podremos (y de hecho es bastante recomendable) almacenar los resultados de forma local para evitar volver a realizar la misma llamada. En el caso de los tweets, uno de los recursos que usualmente recibe mayor número de peticiones, lo óptimo es almacenar todos los tweets recibidos hasta el último, y hacer uso del parámetro *since_id* para recuperar sólo los tweets que no tenemos almacenados en nuestro sistema. Además de ayudar a no sobrepasar el límite conseguiremos no tener que repetir el procesamiento de todos esos tweets que ya tendremos almacenados.

La otra manera es suscribirnos a la API de Streaming, a partir de la cual recibiremos en tiempo real una serie de recursos, según al stream concreto que nos hayamos suscrito. La información que se recibe de esta API no está sujeta a limitaciones de uso, con lo cual es más viable para conexiones persistentes en las cuales se monitorice el estado de dichos streams.

2.3.4. API Streaming

A diferencia de la API REST, la API de Streaming funciona a la inversa. Al realizar una petición de subscripción a un stream, nos convertimos en escuchadores, a la espera de que Twitter nos envíe las actualizaciones pertinentes que afecten al stream seleccionado, evitando una sobrecarga de peticiones que ocurriría si tuviésemos que hacer un polling permanente sobre dichos recursos.

En los siguientes diagramas se observa la diferencia entre las dos APIs:

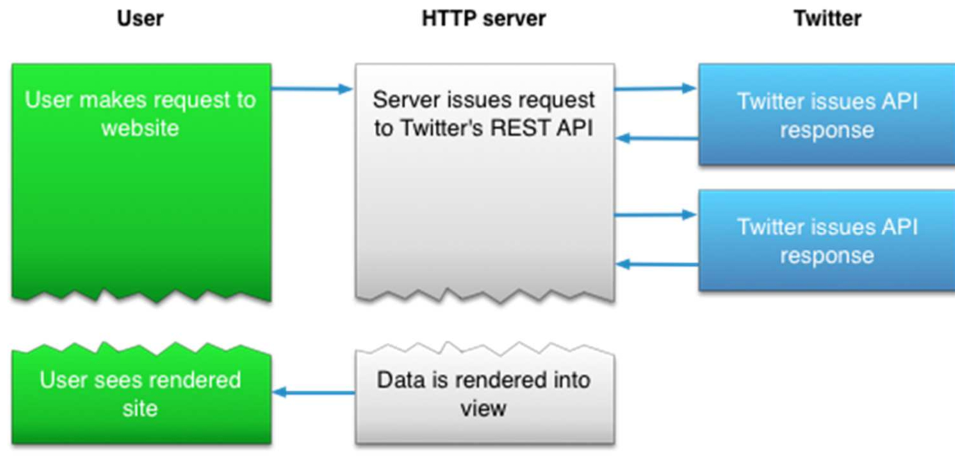


Ilustración 7 – Flujo de conexión en API REST

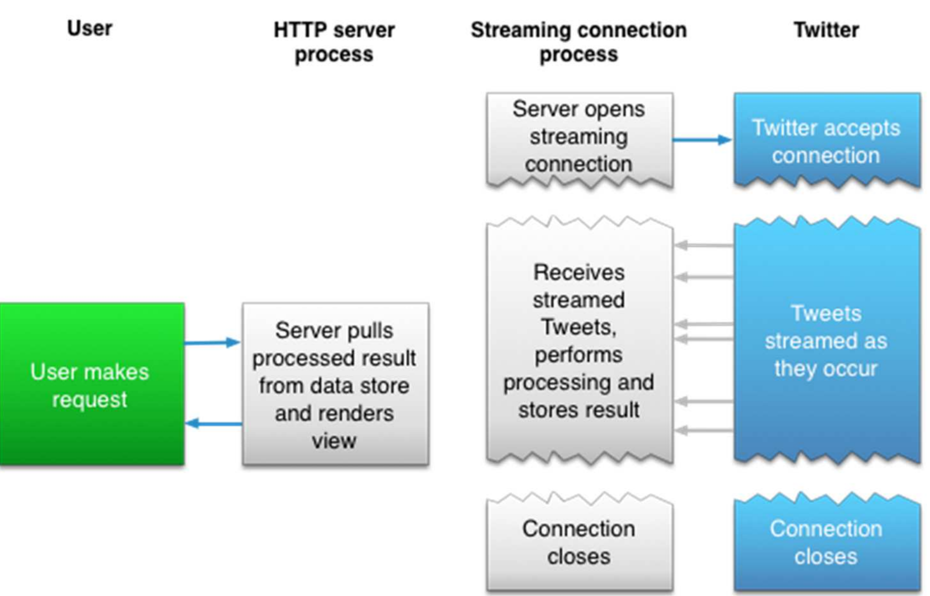


Ilustración 8 – Flujo de conexión en API Streaming

Resumiendo, tras iniciar el stream con la petición a Twitter se crea una conexión HTTP persistente entre Twitter y el servidor de la aplicación, el cual recibirá en tiempo real las actualizaciones de Twitter. De forma paralela, el servidor podrá tener otro tipo de conexiones abiertas, hacia clientes u otros servicios, y aprovechar las informaciones recibidas de Twitter conforme llegan (en el ejemplo se ve una petición HTTP de un usuario final).

Twitter dispone de tres streams a los que poder suscribirse: los streams públicos, que son un muestreo de distintos tweets elegidos al azar de entre todos los usuarios; streams de usuario, que involucran toda la actividad necesaria para el seguimiento de un usuario en particular; y streams de sitio, que todavía están en beta, permiten que un sitio con varios usuarios pueda monitorizar a todos de forma similar a los streams de usuario, especificando en cada actualización a que usuario se refiere.

El stream usado por la aplicación será el stream de usuario, invocado a través de la llamada *GET user* de la API REST, ya que lo que se necesita es acceder a los tweets que recibe específicamente la cuenta de la aplicación del sistema.

Por tanto, las ventajas de utilizar la API de Streaming son obvias, recibimos las actualizaciones de forma automática y en tiempo real, podemos obviar las limitaciones bastante estrictas de la API REST y minimizamos el overhead por múltiples consultas que aparecería si tuviésemos que monitorizar mediante polling los recursos que queremos obtener del usuario (normalmente tweets). La única pega del uso de la API de Streaming es que su uso implica que la aplicación tenga una arquitectura más complicada que con el uso de llamadas simples de REST, pero se compensa con creces con las ventajas, así que es prácticamente obligado hacer uso de los streams cuando se quieren monitorizar los tweets de una cuenta, como en el caso del sistema a implementar en el proyecto.

2.4. Librerías de Google

2.4.1. Introducción

Google ofrece multitud de servicios al público. Todos sus servicios se consumen a través de la web. Aunque Google ha cambiado varias veces su interfaz a la hora de acceder a su API, suele utilizar tecnologías lo más simples posibles y con mayor alcance, todas ellas basadas en la web. Google ha pasado por APIs de servicios web tradicionales (SOAP y/o REST con codificación XML y/o JSON) o APIs para lenguajes específicos (Java o PHP por ejemplo), entre otros.

Sin embargo, a día de hoy la tendencia es a considerar obsoletas o incluso retirar dichas APIs, sustituyéndolas por APIs JavaScript que utilizan AJAX. Podemos encontrar multitud de ejemplos ejecutables de algunas de estas librerías en el *Code Playground* de Google [47].

AJAX es acrónimo de *Asynchronous JavaScript And XML.* En esencia, AJAX es una manera de trabajar sobre la web, consistente en realizar peticiones asíncronas desde el cliente a una página web [48]. Esto permite cargar nueva información mediante llamadas concretas al servidor, las cuales recibiremos de forma concurrente con la ejecución de la página. De esta manera, podremos modificar de forma dinámica la página del lado del cliente realizando consultas al servidor y tratar los resultados con JavaScript, sin tener que estar continuamente recargando la página, lo cual permite tanto mejorar la experiencia de usuario como ejecutar llamadas a servicios web en el contexto de una aplicación web.

En estas APIs suele ser recomendable el uso de un código de aplicación, que se obtiene a través de una cuenta Google del desarrollador [49]. A partir de esta clave Google reconoce la aplicación y envía la petición de la API a sus servidores (al endpoint del servicio web), que procesa la

información y la devuelve, haciendo uso de AJAX para que se refresque la información en el cliente, haciendo parecer que la llamada se ha realizado completamente en el cliente como una llamada JavaScript normal y corriente.

Esta forma de acceder a APIs basadas en servicios web no es ningún estándar (como lo son la Web Service Protocol Stack) ni está soportado de ninguna manera por ninguna organización, pero es una manera de proceder válida y que está ganando muchos adeptos, sobre todo para el desarrollo de aplicaciones web para el cliente.

Aun así, todavía hay algunas APIs de Google, sobre todo si involucran trabajos pesados, que siguen usando el esquema clásico. Un ejemplo serían las APIs auxiliares a Google Maps (por ejemplo los sistemas de enrutado o Google Places) [50].

Con respecto a los servicios de Google en plataformas Android, aunque no voy a entrar en detalle, algunos de los servicios de Google como Google Maps o Google Plus se ofrecen de forma nativa en la API de Android, la cual está escrita en Java, o a través de unas librerías estándar de Android denominadas Google Play Services [51]. En caso de usar alguna de estas APIs en Android sí que está recomendado hacer uso de la oficial, aunque en cualquier caso se podrán usar todos los servicios, incluidos éstos, haciendo uso directamente de las librerías basadas en servicios web, igual que si accediésemos desde un PC convencional.

Por último, es importante comentar que algunos de los servicios web de Google, tales como Google Translate [52] o Google Maps For Business [53], pueden incluir tarifas por su uso, por lo cual habrá que configurar primero su método de pago, y además tener cuidado de hacer un uso lo más escaso posible de dichos recursos, para evitar costes innecesarios.

2.4.2. Google Maps

De todos los servicios de Google, el más relevante para el proyecto es Google Maps, ya que es la base para poder representar sobre un mapa la información de las incidencias obtenidas.

La API de Google Maps [54] incluye todas las opciones que se pueden encontrar en la página de cliente de Google Maps [55] repartidos entre diferentes servicios. El núcleo de esto es una API JavaScript con la cual cargamos un mapa al documento HTML y con la cual accedemos a la mayoría de las funcionalidades del mapa (por ejemplo: movimiento, zoom, añadir y eliminar marcadores, etc.), las cuales se realizan mediante peticiones web a Google, que se reflejan en el mapa de la página utilizando AJAX.

Algunas opciones más avanzadas o complementarias se pueden añadir con el uso de otras APIs auxiliares de Google Maps. Veamos de forma resumida estas opciones:

- Google Maps for Business: Una versión de la API básica con algunas opciones mejoradas y sobre todo mejor soporte y SLA (Service Level Agreement) exclusivo para empresas.
- Google Static Maps: Incrusta una imagen referenciando un determinado mapa en tu página de forma estática (no se puede modificar el mapa mediante JavaScript). Útil para páginas que quieran incluir un mapa fijo. También se puede acceder así a imágenes de Street View.
- Google Places: Para encontrar lugares de interés en algún contexto geográfico. Se puede usar en combinación con la API JavaScript cargando unas librerías adicionales o usar su API propia basada en servicios web tradicionales.
- Otros servicios web: Se ofrecen otros servicios (los más pesados) a partir de servicios web que no se incluyen en la API JavaScript. Estos son: rutas, matriz de distancia, elevación y codificación geográfica.

La API de Google Maps, en su versión gratuita, está restringida a 25000 cargas de mapa diarias (sólo incluye la inicialización, el resto de operaciones no cuentan para el límite). Si se va a dar uso a múltiples usuarios o se van a realizar muchas cargas diarias, es recomendable hacer uso de una clave de desarrollador para la API, fácilmente conseguible de forma gratuita en *https://code.google.com/apis/console* y que se vincula a una cuenta de usuario en Google.

En caso de superar el límite durante varios días consecutivos, se cortará nuestro acceso a la API de Google Maps. Si necesitamos más cargas diarias por tener un alto número de usuarios podemos realizar dos cosas: activar la facturación de la API (desde 25000 hasta 250000 cargas diarias), con la cual nos cobrarán el exceso de cargas de mapas (aquellas por encima de las 25000 gratuitas), o suscribirnos a Google Maps for Business, con el cual a partir de una cuota tendremos un número ilimitado de consultas disponibles. A no ser que el límite se traspase por poco, será menos costosa la suscripción, aunque puede depender del caso concreto.

Para hacer uso de la API JavaScript tendremos que obtenerla a través de la siguiente dirección, directamente en el código de la página:

```
http://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&sensor=SET_TO_TRUE_OR_FALSE
```

En key tendremos que introducir nuestra clave de acceso a la API (opcional) y en sensor lo fijaremos a true o false, según si nuestra aplicación hace seguimiento de la posición o no (obligatorio).

Para realizar una carga completa del mapa en nuestro sitio web tendremos que utilizar el siguiente código HTML/JavaScript, que puede adaptarse ligeramente para nuestras necesidades:

```
<!DOCTYPE html>
<html>
 <head>
  <meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
  <style type="text/css">
   html { height: 100% }
   body { height: 100%; margin: 0; padding: 0 }
   #map_canvas { height: 100% }
  </style>
  <script type="text/javascript”
src="http://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&sens
or=SET_TO_TRUE_OR_FALSE">
  </script>
  <script type="text/javascript">
   function initialize() {
    var mapOptions = {
     center: new google.maps.LatLng(-34.397, 150.644),
     zoom: 8,
     mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    var map = new
google.maps.Map(document.getElementById("map_canvas"),
      mapOptions);
   }
  </script>
 </head>
 <body onload="initialize()">
  <div id="map_canvas" style="width:100%; height:100%"></div>
 </body>
```

```
</html>
```

En este ejemplo se crea un elemento *map_canvas*, el cual ocupa todo el espacio de la pantalla, aunque lo único importante es tener un elemento *div*, que será el que incluya el mapa. En el ejemplo ocupa el 100% de la pantalla, aunque podrá tener cualquier tamaño o posición en la página, y tener cualquier id.

Posteriormente, será necesario cargar la API de Google Maps como se ha dicho anteriormente. Con esta API se crea una función de inicialización, en la que será necesario expresar como mínimo las opciones básicas del mapa: coordenadas del centro del mapa, nivel de zoom y el tipo de mapa. Esta función se llama con la carga de la página, y vinculará la capa HTML con el mapa definido, que después de podrá modificar libremente.

En la web de la API JavaScript de Google Maps podemos encontrar la referencia completa de la API [56], sin embargo, voy a comentar algunas de las clases y métodos que me han resultado más importantes.

Clases:

- **Map**: La clase que contiene toda la información y métodos del mapa.
- **MapOptions**: La clase que especifica las opciones del mapa.
- **Marker**: Un marcador de posición.
- **MarkerOptions**: Las opciones del marcador.
- **Icon**: Un icono para colocar en un marcador.
- **LatLng**: Posición especificada por un par latitud-longitud.
- **LatLngBounds**: Un objeto que especifica unos límites rectangulares en base a las coordenadas de 2 esquinas.
- **Size**: Objeto que especifica el tamaño de un elemento (por ejemplo, un icono).

Métodos:

- **Map.fitBounds(LatLngBounds)**: Posiciona el mapa para que encaje perfectamente con los límites especificados. Se realiza de forma asíncrona y realiza un desplazamiento y un zoom.
- **Map.panTo(LatLng**): Mueve el mapa a una coordenada concreta.
- **Map.panBy(x,y):** Mueve el mapa la distancia especificada (*x* para el eje horizontal e *y* para el vertical).
- **Map.setCenter(LatLng):** Fija un nuevo centro para el mapa, y lo mueve a dicho centro.
- **Map.setZoom(Number)**: Fija un nuevo nivel de zoom.

- **Marker.setMap(Map)**: Fija un marcador a un mapa concreto. Con esto se dibuja el marcador en dicho mapa. Para eliminarlo se puede fijar un mapa nulo (marker.setMap(null)).
- **LatLngBounds.extends(LatLng)**: Extiende unos límites para que incluyan de la forma más mínima posible al nuevo punto especificado.

El uso de la API de Google Maps para Android se puede realizar de forma oficial gracias a la API para Android. Esta API está escrita en Java, aunque es una conversión de la API JavaScript. Los elementos a los que se accede y las funciones que se utilizan y cómo se utilizan es exactamente igual, cambiando únicamente clases JavaScript por clases Java. La única diferencia está en la manera de inicializar los mapas.

Los mapas se crean de forma estática mediante un documento XML, como el resto de elementos de interfaz en Android (veremos más detalles sobre esto en su sección) y luego se puede referenciar accediendo a partir de su id en código. Con esto obtendremos la referencia a un objeto de tipo *GoogleMap*, que es el equivalente al objeto Map en Javascript. A partir de aquí, podremos acceder a los mismos métodos y clases que con el equivalente de la API JavaScript [57].

3. Tecnología Android

3.1. Introducción

Ya que el desarrollo del cliente de la aplicación se ha implementado en base al sistema operativo Android, es importante explicar sus posibilidades y su funcionamiento, así como su posición dentro del mercado actual de los dispositivos inteligentes (smartphones).

El proyecto Android surgió a partir de la creación en 2004 de una empresa, que se hizo llamar Android Inc., cuyo proyecto consistía en crear un sistema operativo propio basado en Linux para dispositivos inteligentes, que permitiese un uso sencillo de las capacidades básicas de estos terminales: uso de pantalla táctil, acceso a datos típicos de un teléfono (contactos, llamadas, etc.), capacidad de acceso a Internet, facilidad en la creación de apps, etc. En esta época los smartphone todavía no habían aparecido como tal, aunque eran un proyecto de futuro importante para compañías del sector como BlackBerry o Nokia, además de interesar a otras como Apple (que sacará en 2007 su popular iPhone) o Google.

En 2005 se produjo la compra de Android Inc. por parte de Google [58], de tal manera que se hizo con el desarrollo del sistema operativo Android, así como los derechos para realizar la venta de teléfonos basados en dicha tecnología. Gracias a las aportaciones de Google y la Handset Open Alliance (formada por muchos de los principales actores del sector, incluyendo a Google, Samsung, Motorola o Intel, entre otros) se terminó de desarrollar la versión 1.0 de Android, que se presentó de forma oficial en 2007 [59]. Posteriormente en 2008 se lanzaría el primer terminal comercial con Android, el HTC Dream.

A partir de aquí comienza una batalla entre los distintos sistemas operativos para smartphones, que sigue en la actualidad, y cada vez más encarnizada debido al mayor número de smartphones vendidos y su increíble popularidad. En un principio teníamos como principales rivales a Android a los siguientes:

- iPhone (iOS)
- Blackberry (BlackBerryOS)
- Windows Mobile/Windows Phone (Windows)
- Nokia (Symbian OS).

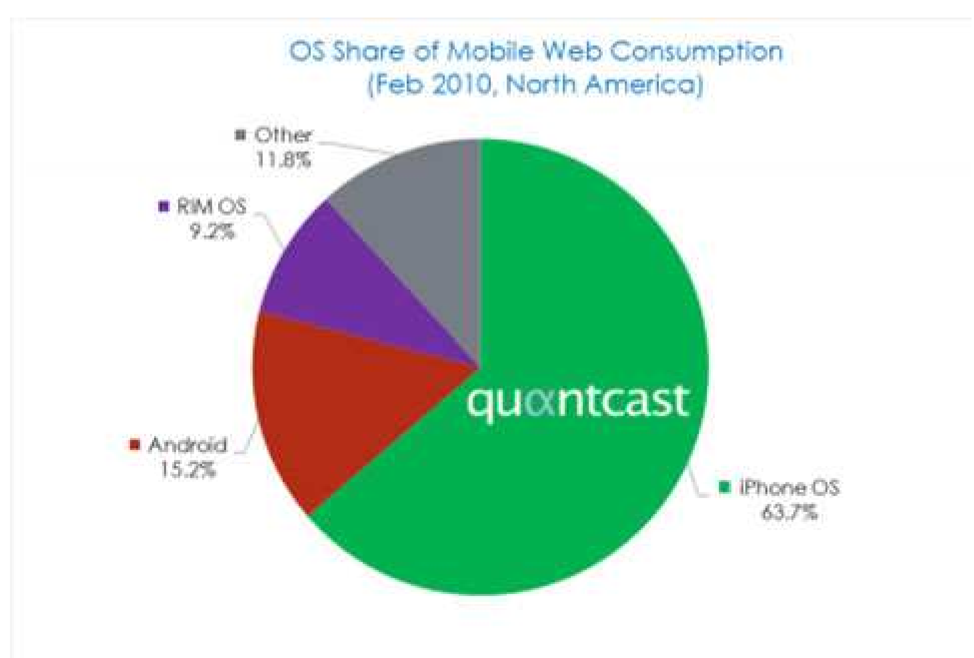


Ilustración 9 – Cuota de mercado de smartphones en 2010

A día de hoy tenemos que los Android se han consolidado enormemente, siendo los más vendidos con diferencia. En segundo lugar tenemos iPhone, a mucha distancia de Android pero todavía con una relevancia considerable. Además, todavía quedan algunos restos, aunque bastante pequeños para Windows Phone y Blackberry.

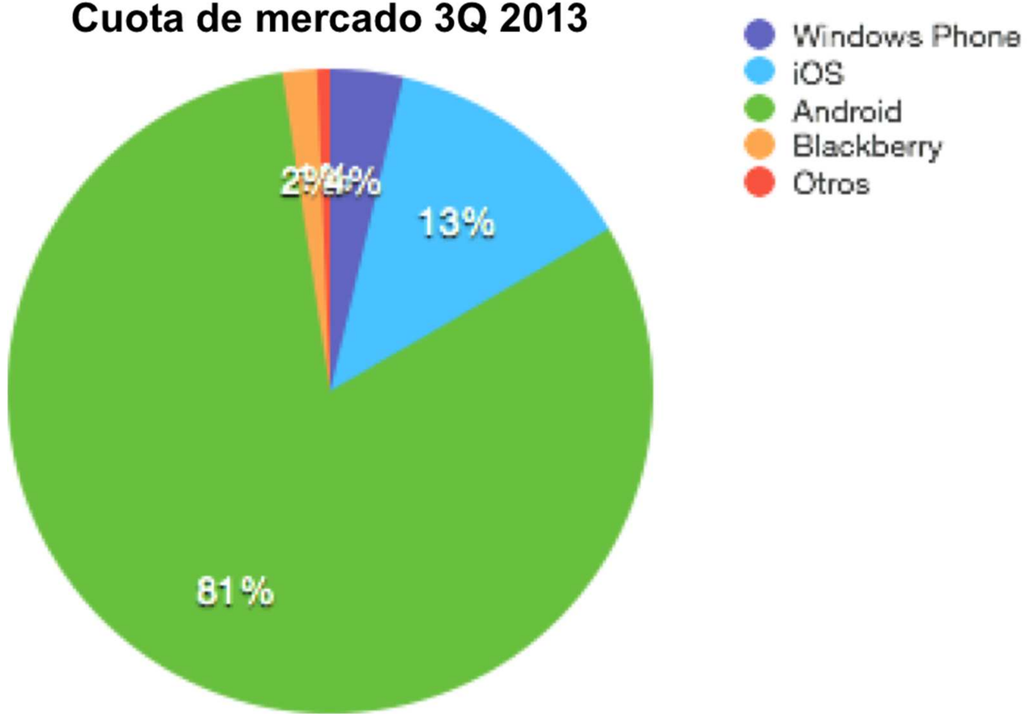


Ilustración 10 – Cuota de mercado de smartphones en 2013

Además de esta cuota de mercado tan amplia, es importante mencionar que el abanico de posibilidades es mayor, teniendo muchos más fabricantes que

hacen muchos terminales distintos, de distintas prestaciones y precios, algo que es más complicado de encontrar para el resto de sistemas operativos.

Pasando a cuestiones más técnicas, Android se ha construido como un sistema operativo desarrollado principalmente en C y Java, basándose en Linux en su mayor parte.

Al principio el sistema operativo era completamente funcional, pero bastante pobre en cuanto a funcionalidades interesantes de cara al usuario. Por tanto, su mejor baza era su sistema abierto (mucho más que el resto de competidores) y la facilidad para el desarrollo de aplicaciones de terceros. Esto es así gracias a una API sencilla y muy potente, basada en Java, y a un mercado de venta de aplicaciones muy abierto y sencillo de cara al desarrollador. Con el tiempo, se ha mantenido dicha sencillez y abertura en el ecosistema, y se han arreglado y mejorado todos los aspectos relacionados con la experiencia de usuario. En las distintas versiones que han ido apareciendo de Android se han añadido características que hoy se consideran esenciales y se ha mejorado la estabilidad del sistema y su seguridad, además de añadir por defecto una serie de funcionalidades a la API que facilitan el desarrollo de aplicaciones de terceros.

Aquí se puede observar una lista de características más importantes de las distintas versiones de Android [60] [61]:

- **Android 1.0**: Soporte básico para las tareas más importantes como mensajería, email, cámara, Internet, aplicaciones de Google (Gmail, Google Talk, Youtube, etc.), navegador web, Android Market, etc.
- **Android 1.6 (Donut)**: Mejoras para la cámara, mejora en la búsqueda, Text-To-Speech, otras mejoras leves.
- **Android 2.0 (Éclair)**: Múltiples mejoras de cámara (flash, zoom digital y otros) y mejoras en múltiples aspectos (multicuenta, HTML5, Salvapantallas dinámicos, Teclados virtuales, etc.)
- **Android 2.2 (Froyo)**: Mejorado el soporte de idiomas y de fotos y videos. Se añade soporte para Flash y la instalación de aplicaciones en tarjeta SD.
- **Android 2.3 (Gingerbread)**: Soporte para VOIP, NFC y para múltiples cámaras. Se añade un gestor de descargas y de energía.
- **Android 3.0 (Honeycomb)**: Mejoras en la interfaz de usuario (teclado, aplicaciones de contactos y email o galería, entre otros). Adaptación al tamaño de pantalla más grande de las tablets. También incluye cambios a la hora de crear interfaces y widgets.
- **Android 4.0 (Ice Cream Sandwich)**: Soporte para Speech-To-Text y para compartir datos mediante NFC. Mejoras de usabilidad en mensajes y emails. Se mejoran múltiples características de la cámara y se añade reconocimiento de caras. Se realizan mejoras importantes en el navegador

web. Mejoras varias en la interfaz y la usabilidad. Sistema adaptado tanto para tablets como para móviles.

- **Android 4.1 (Jelly Bean)**: Mejoras para la alta resolución. Opciones de accesibilidad para ciegos. Aumento de los lenguajes soportados. Mejoras de rendimiento del software de la cámara y del navegador web. Mejoras varias en los widgets. Mejoras de rendimiento en la interfaz de usuario. Aparición del sistema inteligente Google Now.
- **Android 4.4 (KitKat)**: Rediseño de la interfaz de usuario. Búsquedas ampliadas en la agenda. Integración de sistemas de mensajería. Mejoras varias de rendimiento. Emulación de NFC. Optimizaciones de consumo de batería.

3.2. Esquema básico de una aplicación

3.2.1. Actividades

Pasaré ahora a explicar la estructura básica para el desarrollo de una aplicación en Android, cuáles son sus partes fundamentales y como se estructura y se desarrolla la implementación de dicha aplicación.

El primer concepto con el que nos encontramos al empezar el desarrollo de una app, es el concepto de actividad. Tomando la definición de la propia documentación de Android [62] encontramos que una actividad (o *Activity*, en inglés) es cada uno de los elementos únicos en los que un usuario puede trabajar a la vez. Por tanto es la unidad en la cual se dividen los programas de manera lógica en Android.

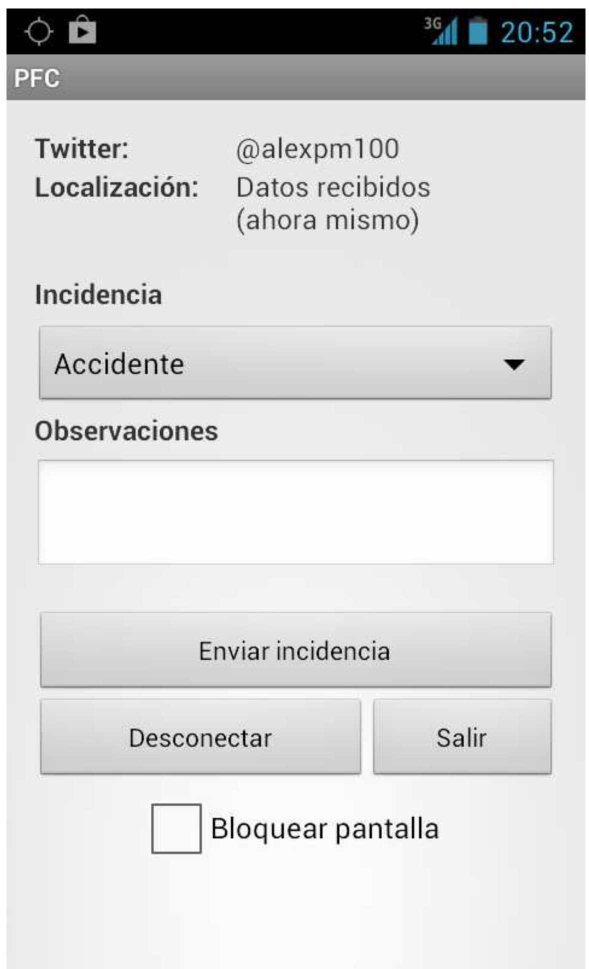


Ilustración 11 – Captura de pantalla de una Actividad en Android

Normalmente estas actividades corresponden con una pantalla completa del smartphone, y se relacionan de forma directa con una sección importante de la interfaz de usuario, aunque puede ocurrir tanto que no se disponga de interfaz como que sea un diálogo modal a otra actividad padre. Los casos en los que no se dispone de interfaz suelen ser actividades que realizan alguna operación en segundo plano, que aunque pueda ser importante para el usuario no requiere de interacción directa con él.

Todas las aplicaciones deben constar como mínimo de una actividad, denominada actividad principal, que es la que se lanzará por defecto al inicio de la aplicación. Esta actividad suele ser la pantalla de inicio de la aplicación, aunque puede ser una actividad previa, tal como una pantalla de login o una "splash screen". Además de la actividad principal, una aplicación puede incluir cuantas actividades necesite.

Para la creación de las interfaces de usuario que se mostrarán en las actividades se hace uso de una herramienta llamada layouts. Los layouts, de forma similar a otros lenguajes de programación o frameworks, es la manera de realizar la maquetación de los distintos elementos gráficos de la interfaz de usuario, posicionándolos de forma correcta en la pantalla del

dispositivo. Como suele ser habitual en este tipo de herramientas, permite la capacidad de realizar diseños que se adapten a distintos tamaños de pantalla, resoluciones, dispositivos, orientación de la pantalla, etc. De hecho, es lo más recomendable para tener un correcto visionado en distintos dispositivos.

Estos layouts se crean como un recurso, escrito en XML, en el cual se introducen los distintos elementos (denominados vistas, o *View*) de forma jerárquica, con sus atributos determinados. Algunos de estos elementos más importantes son:

- **LinearLayout**: Se utiliza para posicionar elementos en línea recta, que puede ser de forma horizontal o vertical. Además permite especificar pesos a los distintos elementos, para que ocupen más o menos espacio dentro del layout.
- **TableLayout**: Posiciona sus elementos en forma de tabla, en la cual especificaremos cuantas filas y cuantas columnas disponemos.
- **TextView**: Etiqueta de texto, de forma similar a un Label en otros lenguajes.
- **EditText**: Un texto editable, puede ser de línea única o múltiple.
- **Button**: Un botón al que se puede asociar un evento que se lanza cuando se pulsa el botón.
- **Spinner**: Lista desplegable para elegir una opción única de entre varias opciones.
- **CheckBox**: Un elemento que permite dos estados: seleccionado o no seleccionado.

Podemos ver a continuación un ejemplo de cómo quedaría la definición en XML de un layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical" >
   <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
```

```
  <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

Cuando creamos una actividad, tenemos que especificar el layout que queremos que se asocie a dicha actividad. El layout especificado pasará entonces a conformar la interfaz de usuario de esa actividad, aunque podrá ser modificada posteriormente. Si no se especifica layout se obtendrá una actividad sin interfaz.

3.2.2. Ciclo de vida

Como se puede sacar de su definición, la característica más importante de una actividad es que sólo puede haber una actividad ejecutándose en cada momento. De esta manera, será muy importante gestionar los cambios de estado de una actividad, y como afecta esto a su desarrollo. Esto es lo que se conoce como ciclo de vida de una actividad.

Una actividad puede pasar por cuatro estados fundamentales:

- Activa: Es el estado básico de una actividad, se da cuando la aplicación está activa en pantalla y ejecutándose.
- Pausada: Ocurre cuando la actividad está en pantalla pero no tiene foco y por tanto no se está ejecutando. Puede ocurrir cuando alguna actividad modal o diálogo se lanza y pausa la actividad activa, o cuando algún proceso requiere atención mientras la actividad está activa pero no la manda a segundo plano.
- Parada: La actividad sigue existiendo y retiene sus estados en memoria, pero está completamente oculta por otra actividad y no se presenta en pantalla. Puede ser restaurada a primer plano en cualquier momento por el usuario o por algún efecto de la aplicación.
- Finalizada: Esto ocurre cuando el usuario termina de forma definitiva el uso de la actividad, o cuando la aplicación decide que ya no es necesaria, o cuando el sistema operativo decide limpiarla para recuperar su memoria.

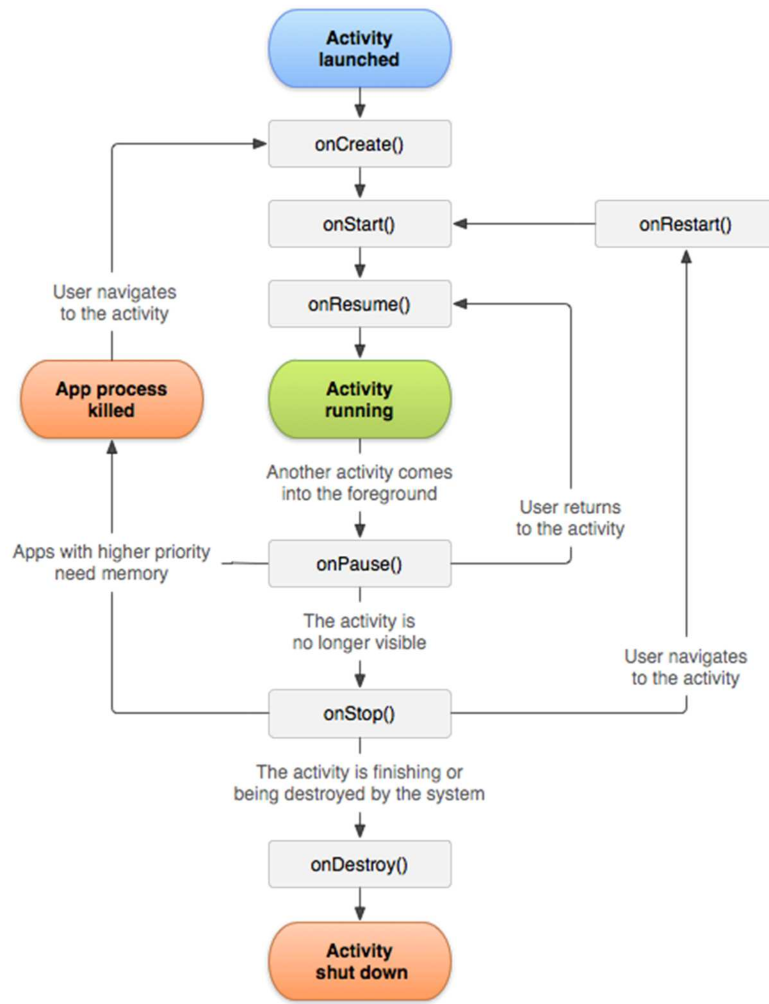


Ilustración 12 – Ciclo de vida de una actividad en Android

En el esquema superior se puede observar el paso de los distintos estados, y qué eventos se lanzan en sus transiciones, los cuales corresponden a distintos conceptos básicos del ciclo de vida de la actividad.

Todos estos eventos son implementables por la aplicación, para personalizar su comportamiento en dichos casos. De hecho, esto suele ser recomendable en la mayoría de ocasiones, para poder realizar tareas secundarias o mantener el estado en diferentes eventos.

Para la creación de una actividad el método más importante a implementar es onCreate(). Este método se llama para crear la actividad y es el único cuya implementación es obligatoria. En él se carga la interfaz de usuario (en caso de haberla) de su fichero XML y se inician todos los recursos de la actividad. Como contrapunto, al finalizar de forma definitiva la aplicación se lanzará onDestroy(), por tanto habrá que aprovecharlo para limpiar todos los recursos que puedan suponer un consumo importante (conexiones a Internet, servicios complejos como el posicionamiento o el reconocimiento de voz, etc.)

Así mismo, cuando la actividad pase a segundo plano, será importante hacer uso de onPause() para parar los recursos pesados (pero no eliminarlos del todo) y onResume() para volver a iniciarlos.

Finalmente, es importante considerar que mientras la actividad no sea eliminada se almacena su estado de interfaz de forma automática, el cual es restaurado al volver a primer plano en la pantalla del smartphone. Sin embargo, para recursos que no son de la interfaz (cualquier variable miembro de la actividad, por ejemplo) será necesario guardar y recuperar de manera manual dicha información en los métodos correspondientes.

3.2.3. Recursos

Muchos de los componentes de una aplicación Android suelen incluirse en forma de recursos. Estos son archivos estáticos, la mayoría en formato XML, que incluyen algún contenido importante de la aplicación, como la interfaz de usuario o las cadenas de texto.

La separación entre código y recursos es muy importante en el ámbito del desarrollo de aplicaciones, ya que permite encapsular todos los datos que incluye la aplicación en archivos separados y fácilmente modificables. Esto simplifica enormemente los cambios y modificaciones a los recursos, además de reducir el tamaño del código de relleno dentro de los archivos de código fuente de la aplicación.

Una característica fundamental y muy útil dentro de los recursos en Android es que permiten de forma sencilla el uso de recursos alternativos. Con recursos alternativos podemos tener un recurso especificado de distintas maneras para distintos casos. Ejemplos básicos de esto se pueden encontrar en la internacionalización de cadenas de texto (un archivo para cada idioma) o en la adaptación de la interfaz para pantallas de distintas orientaciones o tamaños (en el caso más extremo incluso smartphones y tablets). Android siempre elegirá el recurso más adaptado de forma automática, en el caso de que haya varios recursos alternativos para el mismo tipo de recurso.

Los recursos se encuentran siempre dentro de la carpeta *res* del proyecto de la aplicación, y cada tipo de recurso se encuentra en una carpeta diferente. El nombre de la carpeta es la que determina el tipo de recurso, siguiendo la siguiente especificación:

```
<nombre_recurso>[-<modificador>]*
```

Algunos ejemplos de carpetas de recursos serían: drawable, layout-land-large o values-es.

El nombre del recurso especifica el tipo de recursos que se guardan en la carpeta, y es el único elemento obligatorio. En el caso de que no se incluyan modificadores se considera que es el recurso por defecto para

ese tipo de recursos, y muchas veces ocurrirá que es el único recurso de ese tipo (en aquellos casos en los que no se usan recursos alternativos).

Una lista de los recursos más importantes que se pueden utilizar sería la siguiente:

- **drawable**: Colección de imágenes que se pueden dibujar en la aplicación
- **layout**: Archivos en XML que especifican la maquetación de la interfaz de usuario.
- **menu**: Especificación en XML de los menús de la aplicación.
- **raw**: En esta carpeta se puede incluir cualquier tipo de archivo en formato binario, que luego se podrá leer como recurso en la aplicación.
- **values**: Se utiliza para especificar listas estáticas de valores. Los archivos más usados para este tipo de recursos son las cadenas de texto (strings.xml), aunque también pueden describirse colores, arrays de valores, dimensiones o estilos (de interfaz, similar a un archivo CSS pero en XML).
- **xml**: Carpeta para almacenar recursos arbitrarios en XML, que podrán ser leídos por la aplicación.

Los modificadores de los tipos de recurso se utilizan para especificar casos concretos para los recursos alternativos. Si no se especifica ninguno será el caso por defecto. Se pueden especificar un número arbitrario de modificadores para una carpeta de recursos, pero lo más habitual es especificar sólo uno. Los modificadores más comunes son los siguientes:

- **Lenguaje y región**: Se utilizan para la internacionalización, sobre todo en el caso del archivo de cadenas strings.xml y de forma general en los archivos que requieran una localización específica. Para el lenguaje se utilizan los códigos de dos letras de ISO 639-1 (por ejemplo es ,en, fr). De forma opcional se puede añadir la región especifica siguiendo el formato de ISO 3166-1-alpha-2 (ej: fr-rFR, fr-rCA)
- **Orientación de escritura**: Para especificar diferentes layouts para paises que escriben de izquierda a derecha o de derecha a izquierda. Se usan los códigos ldltr y ldrtl.
- **Tamaño de pantalla**: Se utiliza para especificar layouts distintos para adaptarse a distintos tamaños de pantalla. Se pueden utilizar cuatro valores predeterminados: small (320x426 dp), normal (320x470 dp), large (480x640 dp) y xlarge (720x960 dp, generalmente tablets).

- **Orientación de pantalla**: Se usa para mostrar distintos layout cuando la pantalla está horizontal y cuando está vertical. Los valores son port (vertical, por defecto) y land (horizontal).
- **Densidad de píxeles**: La densidad de píxeles es una relación entre la resolución y el tamaño real de la pantalla, y se mide en dpi (dots per inch, o puntos por pulgada). Esto hace que las imágenes se tengan que escalar para que se adapten a las distintas densidades. Por ello se utiliza dentro de los recursos de imágenes para poder incluir imágenes de distintas resoluciones que se adapten mejor a distintas densidades. Los valores permitidos son: ldpi (baja densidad: 120 dpi), mdpi (media densidad, 160 dpi), hdpi (alta densidad, 240 dpi), xhdpi (extra alta densidad, 320 dpi), nodpi (imágenes fijas que no queremos que se escalen) y tvdpi (para aplicaciones que se ejecutan en TV, unos 213 dpi).
- **Versión de Android**: Podremos especificar distintos recursos para distintas versiones de Android, esto es útil para poder usar elementos de versiones nuevas sólo en versiones nuevas y usar los viejos en versiones anteriores. Se nombra como vN, donde N es la versión de la API de Android (ej: v1, v7, v10).

Para hacer uso de los recursos de la aplicación Android compila de forma estática los archivos de recursos, creando una clase que no se debe tocar, llamada *R*. Podremos acceder a los identificadores de distintos elementos de los recursos desde cualquier parte de la aplicación usando la siguiente sintaxis Java:

```
R.<tipo_recurso>.<nombre_recurso>
```

Algunos ejemplos podrían ser R.string.appName, R.color.blue o R.layout.main.

Generalmente, acceder directamente a los identificadores no sirve para nada, ya que son exclusivamente un número para referenciar el recurso. Por tanto, existen una serie de métodos por defecto para acceder a la información de los recursos a partir de su identificador. Algunos ejemplos serían *findViewById* para obtener elementos de la interfaz o *getString* para devolver la cadena de un recurso de texto.

3.2.4. Comunicación

Uno de los aspectos más importantes dentro del desarrollo para Android y dispositivos inteligentes en general es la importancia de la comunicación dentro del sistema. Esta comunicación se dará a múltiples niveles: encontramos intercambios de información entre actividades, aplicaciones, servicios, sistemas, etc. Accedemos a servicios externos y a Internet, y es importante que todos estos elementos se comuniquen de forma correcta entre ellos. Esto suele traer como problema un cambio en el paradigma

típico de programación en el cual tenemos una aplicación local con un flujo único que seguir.

Android tiene muchos mecanismos integrados en su API para resolver estos problemas e integrar la comunicación entre componentes de una aplicación y entre aplicaciones distintas de forma sencilla. Vamos a ver ahora algunos de los más importantes.

El elemento clave y más importante para la comunicación es el llamado *Intent*, que permite la comunicación entre distintas actividades. De forma general, un Intent es un "intento" o "intención" de una actividad de realizar una acción, que en ocasiones puede fallar. Un Intent se puede realizar para varias acciones, aunque su función principal es la de comunicar dos actividades entre ellas.

Hay varias maneras de realizar esta comunicación con Intents. La más sencilla es la de simplemente invocar una actividad nueva para que comience y se coloque como la nueva actividad visible en pantalla. Esto se hace llamando a *startActivity(Intent)* con un Intent que referencie a la nueva actividad a empezar. En el caso de que queramos pasar parámetros necesarios para la inicialización de la nueva actividad, podremos introducirlo como datos extra del Intent, con la función *putExtra(String,Bundle)* en la que se especifican la etiqueta de los datos y los datos en sí.

A la hora de comenzar una nueva actividad con un Intent, también podremos lanzar esta actividad esperando un resultado de vuelta a la actividad de origen. Esto se hace usando *startActivityForResult(Intent,int)* en el cual debemos especificar un código de petición compartido entre la actividad que llama y la que es llamada. La actividad de destino deberá rellenar los datos de retorno mediante *setResult()* antes de finalizar su ejecución. La actividad de origen podrá recoger los datos de vuelta al lanzarse el evento *onActivityResult(),* con un código de petición igual al usado al inicio de la petición.

Un Intent también puede usarse para llamar actividades de aplicaciones externas a la aplicación de origen. Para ello, es necesario que la aplicación de destino especifique los casos en los cuales permite que una aplicación externa se conecte a ella mediante un Intent. Esto lo podemos conseguir mediante los *Intent Filters*, que se declaran junto con el resto de componentes de la aplicación.

Un uso interesante para la comunicación mediante Intents es el uso de los llamados Intent implícitos. A diferencia de los Intent anteriores, que llamaremos explícitos, en vez de llamar de forma directa a una actividad para que se inicie, llamaremos de forma directa a una acción o una funcionalidad, de nuestra aplicación o de otra externa (especificada entonces dentro de sus Intent Filters). Android trae de forma integrada muchas acciones de utilidad a ser llamadas mediante Intent como parte

de sus aplicaciones por defecto. Algunas de ellas son: iniciar una llamada de teléfono, ver un mapa, abrir una página web, enviar un email, buscar información de contactos, buscar información de calendario. Para hacer uso de acciones con permisos de una aplicación externa necesitaremos incluir los mismos permisos dentro de nuestra aplicación, o recibiremos un error de seguridad.

Un elemento clave en muchas aplicaciones son los servicios. Un servicio en Android es un proceso persistente, que permanece en segundo plano y escucha peticiones de otros servicios o actividades. Generalmente esto se realiza para aislar procesos de alta carga computacional separándolos de actividades de usuario (más ligeras) o para centralizar el control de una parte de la aplicación que sea común a varias actividades, o que deba poder ser llamada desde el exterior. Por supuesto, también podemos usar servicios en segundo plano de aplicaciones externas, en el caso de no necesitar implementar nuestros propios servicios. Algunos ejemplos de servicios pueden ser: comunicaciones por red, reproducción de música o video, comunicación entre procesos u operaciones sobre archivos.

Hay dos etapas principales en el ciclo de vida de un servicio: la creación del servicio y el uso del servicio. Un servicio se crea una única vez, por parte de la aplicación padre, generalmente cuando se va a comenzar a usar. A partir de aquí el servicio está creado y puede ser usado por cualquier actividad que disponga permiso para ello, por medio de un Intent. Cada una de las peticiones deberá ser procesada por separado. Para ello tenemos dos opciones: podemos utilizar la clase *Service* e implementar a mano un pool de conexiones entrantes junto con su instrumentación, o podremos utilizar la clase *IntentService*, que se encarga de realizar ese tipo de trabajo por nosotros, además de otras ayudas, como separar el trabajo en hilos diferentes del principal (para encapsularlo en caso de fallo). La mayoría de casos el IntentService será la mejor opción, ya que disminuye el esfuerzo necesario y da lugar a implementaciones con mejor rendimiento. No obstante, habrá algunos casos es los que puede ser necesario un control más exhaustivo de lo que ocurre en el servicio, y para ello necesitemos usar un Service básico.

Por último, una vez que la aplicación que contiene al servicio termine, lo más habitual es cerrar el servicio y denegar al acceso al resto de aplicaciones. Aun así, pueden darse casos en los que el propósito principal de la aplicación sea dar ese servicio a aplicaciones externas, y por tanto no se cierre el servicio, o incluso que se permita que los Intent externos accedan a la propia creación del servicio.

El último mecanismo para comunicar entre actividades y/o aplicaciones que voy a comentar es el *BroadcastReceiver*. Mediante un Intent podemos hacer uso de la función *sendBroadcast()* para enviar un broadcast a todos los recibidores que estén registrados y tengan permisos para acceder a dicho Intent. Todos los BroadcastReceiver registrados escuchando a

dichos Intent (sean de la aplicación emisora o de otra distinta) recibirán la información de forma no bloqueante mediante un evento, de manera que podrán procesar el mensaje de manera oportuna. Es importante destacar que es importante reducir el ámbito del mensaje y los permisos al mínimo, ya que al hacer un broadcast enviamos la información a todas las aplicaciones que estén escuchando, y por tanto sería fácil para una aplicación maliciosa acceder a datos privados enviados mediante broadcast.

3.2.5. Concurrencia

Uno de los pilares dentro del sistema operativo Android es la manera en la cual gestiona aplicaciones y procesos. El sistema operativo es multitarea y altamente concurrente. Todos los procesos y aplicaciones del sistema (incluidas todas aquellas que están paradas o en segundo plano) se ejecutan de forma concurrente, con altos cambios de contexto entre aplicaciones, actividades e hilos de ejecución.

Esto se deja ver también en el paradigma de programación de Android. En Android no existen programas principales o *main* ni flujos únicos de ejecución. La programación en Android está basada en eventos (de sistema o de usuario) y en la ejecución concurrente de múltiples hilos. Esto se ha podido observar ya al ver que el desarrollo de una actividad se basa en la implementación de una serie de funciones que se disparan al ocurrir una serie de eventos relacionados con el ciclo de vida de la actividad (onCreate, onPause, onResume, etc.) aunque, de hecho, todo el sistema y toda su API está desarrollada a partir de estas mismas ideas, que a la larga incurren en una mejora de cara al usuario, aunque al principio puedan parecer más complejas de cara al desarrollador.

Siguiendo un modelo similar al de Java, la ejecución de los programas se realiza dividida en procesos e hilos (*threads*). Cada aplicación (sea de usuario o de sistema) se ejecutará en único proceso, ejecutándose todos ellos a la vez de forma concurrente. En esta capa es el propio sistema operativo el que lo gestiona todo y por parte de los desarrolladores será totalmente transparente. Dentro de cada uno de los procesos la ejecución se dividirá en hilos. En este caso la concurrencia ocurre a un nivel más fino, es una concurrencia a nivel de proceso. Sin embargo ésta es la unidad de ejecución que puede controlar el programador, y por tanto será aquí donde deberemos poner el foco.

Todas las aplicaciones tienen en principio un hilo único, que es controlado por el propio sistema y no podemos controlar. Este hilo es denominado hilo principal, y es el que gestiona la actividad activa en cada momento de nuestra aplicación, incluyendo tanto las tareas de computación como las tareas de interfaz, como el renderizado de los distintos elementos gráficos o la interacción con el usuario.

El hilo principal debe ser siempre lo más ligero posible, pues cualquier ralentización implicará a todas las partes de la aplicación, al ser el que se está ejecutando constantemente. Además, será un perjuicio importante de cara al usuario, que notará que la aplicación es lenta y parece no responder bien a sus acciones. Por ello Android incluye una restricción importante sobre el hilo principal: no se permiten tareas de ningún tipo que impliquen un tiempo de ejecución de más de 5 segundos. En el caso de que esto ocurra, Android parará la aplicación e informará de un error de "Aplicación no responde". Además de esto, a partir de la versión 3.0 de Android, se prohíben expresamente (mediante el lanzamiento de una excepción que termina la ejecución de la aplicación) todas las operaciones de red (que generalmente son las más pesadas) en el hilo principal, por lo cual, será más importante todavía buscar mecanismos que nos permitan usar de forma correcta los hilos y la concurrencia para ofrecer a nuestras aplicaciones Android mayor agilidad en su ejecución.

La forma principal y más básica de ejecutar hilos paralelos al hilo principal es utilizar los Thread de Java. En Android funcionan de manera exactamente igual a las versiones estándar de Java, por lo cual no es necesario estudiar una nueva manera de realizar estos Thread, y por ello es una forma sencilla de adaptar nuestro código a hilos si ya sabíamos realizarlo con anterioridad. Concretamente, lo que se realiza es crear un objeto *Runnable*, en el cual se implementará un método *run()* con el código que deseamos que se ejecute en un nuevo hilo. Posteriormente se crea un Thread que contenga a dicho Runnable y lo ejecutamos haciendo uso de la función *start().*

No obstante, la API de Android ofrece un mecanismo para incluir tareas concurrentes al hilo principal de una forma mucho más simple y directa. Este mecanismo de llama *AsyncTask* (tarea asíncrona) y facilita mucho la inclusión de tareas pesadas dentro del hilo principal, ya que esta clase se encarga automáticamente de la creación y gestión del hilo necesario para que la porción de código pesada que incluyamos se ejecute de forma asíncrona y concurrente a la ejecución del hilo principal de interfaz.

Una tarea asíncrona tiene cuatro pasos principales, que se implementan como funciones de una interfaz, cada una realiza una función concreta dentro de la ejecución de la tarea. Éstas son:

- onPreExecute() : Una función opcional, que puede implementarse para realizar alguna tarea de preparación previa a la ejecución del código principal de la tarea. Principalmente se usa para dibujar algún elemento de interfaz para comunicar el inicio de la tarea (por ejemplo, un diálogo de "espere" o una barra de progreso).
- doInBackground(Params...) : Es la operación en sí, que se ejecutará de forma asíncrona en segundo plano. Se pueden incluir, de forma opcional, uno o varios parámetros a la ejecución. Es la única función que es obligatorio implementar.

- onProgressUpdate(Progress...) : Si necesitamos informar periódicamente del progreso de la operación (sobre todo en el caso de operaciones especialmente largas) se podrá llamar a esta función para informar a la interfaz del progreso. Se suele usar en combinación con una barra de progreso.
- onPostExecute(Result) : Función que se llama cuando la ejecución de doInBackground ha completado. Es opcional y se suele usar para informar en la interfaz de que ha terminado la operación (cerrar un diálogo o completar la barra de progreso, entre otros).

Un ejemplo de uso real de un AsyncTask sería el siguiente:

```
private class EnviarTweetTask extends
   AsyncTask<String, Void, Integer> {

            /**
             * Si el tweet se envía correctamente.
             */
            private static final int SUCCESS = 0;
            /**
             * Si se han excedido el límite de peticiones a Twitter.
             */
            private static final int LIMIT_EXCEEDED = 1;
            /**
             * Si ha ocurrido algún otro error al enviar el Tweet.
             */
            private static final int OTHER_ERROR = 100;

            /**
             * Puntero al boton de enviar tweet.
             */
            private Button boton;

            /**
             * Creamos la nueva AsyncTask para enviar un tweet.
```

```
 */
public EnviarTweetTask() {
	boton = (Button)findViewById(R.id.botonTweet);
}

@Override
protected Integer doInBackground(String... params) {
	try {
		capa.enviar(params[0]);
		return SUCCESS;
	} catch (TwitterException e) {
		if (e.exceededRateLimitation()) {
			return LIMIT_EXCEEDED;
		}
		else {
			return OTHER_ERROR;
		}
	}
}

@Override
protected void onPreExecute() {
	boton.setEnabled(false);
	twitterDialog.show();
}

@Override
protected void onPostExecute(Integer status) {
	twitterDialog.dismiss();
	boton.setEnabled(true);
```

```
                int idMensaje = R.string.tweetError;
                switch(status) {
                case SUCCESS:
                        idMensaje = R.string.tweetExito;
                        break;
                case LIMIT_EXCEEDED:
                        idMensaje = R.string.tweetLimiteExcedido;
                        break;
                case OTHER_ERROR:
                        idMensaje = R.string.tweetError;
                        break;
                }
                Toast toast = Toast.makeText(getApplicationContext(),
                                idMensaje,Toast.LENGTH_LONG);
        toast.show();
        }


}
```

En el ejemplo, que sirve para enviar tweets de forma asíncrona, se ve como obtenemos un botón de forma estática la primera vez que se crea el AsyncTask, así como la definición de una serie de códigos de retorno de la operación. Cada vez que se lanza la ejecución de la tarea asíncrona se lanzará la función onPreExecute(), que en este caso deshabilitará el botón (para no poder mandar tweets mientras se está enviando el actual) y mostrará un diálogo avisando que la operación está en proceso.

Después se ejecutará doInBackground, que realiza el trabajo en sí, mediante las librerías de Twitter usadas en la aplicación. Se utiliza como parámetro una cadena de texto con el mensaje del tweet en cuestión. Al finalizar se devuelve el código de status correspondiente al resultado de la operación. Como el diálogo que se muestra no incluye una barra de progreso, se ha decidido no usar la función onProgressUpdate, así que al terminar la ejecución se lanza directamente onPostExecute, con el código de retorno de la función anterior. En este caso devolvemos el estado de la interfaz al normal (activamos botón y eliminamos el dialogo) y mostramos un mensaje especificando si la operación ha tenido éxito o no.

3.3. Acceso a la localización

3.3.1. Fuentes de localización

Una de las partes clave de la aplicación del proyecto es la obtención de la localización, para poder geoposicionar el mensaje con la incidencia. La obtención de la localización es un proceso teóricamente bastante complejo, aunque en la práctica es bastante sencillo gracias a las librerías y sistemas incluidos en Android para trabajar con el GPS u otras fuentes de localización.

Para obtener la localización en Android (o en cualquier otro dispositivo móvil) tenemos principalmente 3 fuentes para obtener la localización. Veamos cuáles son y cuáles son sus principales características.

Fuente	Precisión	Tiempo de establecimiento	Tiempo de actualización	Permisos
GPS	Muy alta (unos 15 m)	Muy alto (entre 1 minuto y 1 hora)	Muy bajo (menos de 100 ms)	Access Fine Location
Red WiFi	Media / Baja (entre 50 y 200 m)	Bajo / Ninguno (sólo es necesario si no estás conectado, unos segundos)	Muy bajo (menos de 100 ms)	Access Coarse Location
GSM Cell ID	Muy baja (unos 1000 m)	Ninguno	Muy bajo (menos de 100 ms)	Access Coarse Location

Tabla 2 – Fuentes de localización

Analicemos brevemente las distintas tecnologías utilizadas para obtener la localización. La tecnología más usada y la más famosa es, por mucho, GPS. Las iniciales GPS significan Global Positioning System (Sistema de Posicionamiento Global), y fue desarrollado en 1973 por el Departamento de Defensa de los EEUU, siendo por tanto una tecnología militar, aunque se permite el acceso civil a la información que provee [63].

GPS consiste en una serie de satélites (31 desde 2010), los cuales envían periódicamente sus datos de localización y tiempo (según un reloj atómico interno) a la Tierra. Desde la tierra, un receptor GPS puede leer estas señales y realizar un proceso de triangulación para determinar su latitud, longitud y elevación. Normalmente, desde cualquier punto de la Tierra tenemos visibilidad hacia unos 9 satélites, aunque depende de la localización, orbitas de los satélites u otros factores medioambientales puede ser una cantidad variable. No obstante, sólo necesitamos de 4 satélites simultáneos para determinar nuestra posición, aunque a mayor número de satélites se pueden hacer cálculos correctores para incrementar la precisión de la medida.

La razón de necesitar 4 satélites es que se requiere uno para cada una de las variables a obtener, siendo 3 de ellas la localización espacial (latitud, longitud y altitud) y la cuarta el tiempo. Aunque a priori podría no parecer necesario se requiere una sincronización absoluta entre los relojes de todos los dispositivos. Esto se consigue fácilmente en los satélites, al disponer de relojes atómicos sincronizados, pero en el dispositivo receptor es inviable disponer de un reloj atómico por su alto precio, y por eso se realiza a partir de los datos recibidos para realizar la triangulación.

Como ya hemos visto en la tabla, la precisión final de GPS es bastante alta, aunque para muchos de sus posibles usos podría ser un poco baja. Por ello existen varias tecnologías adicionales para mejorar la precisión a la hora de determinar la localización. Lo primero con respecto a este tema es importante comentar que, al ser una tecnología militar, se incluía una interferencia a la señal, denominada "Selective Availability" o SA, que podía llevar el error hasta los 100 m. Debido a que cada vez había más medios para corregir ese error mediante algunas técnicas (en concreto el GPS Diferencial) y al auge cada vez mayor del uso civil de la tecnología GPS, en 2000 la señal SA fue eliminada de la transmisión de los satélites [64].

Con respecto a tecnologías usadas para mejorar la precisión tenemos a WAAS (Wide Area Augmentation System) [65]. WAAS consiste en utilizar unas estaciones de control y unos satélites geoestacionarios para poder realizar cálculos correctivos sobre el error obtenido en las mediciones, haciendo así que se tengan valores mucho mejores, con una precisión en torno a unos 6 metros, en el peor de los casos.

Otra tecnología muy usada para incrementar la precisión de las señales GPS es el llamado GPS diferencial (o DGPS). Este se basa en la toma de datos diferenciales con respecto a unas estaciones terrestres que permitan corregir los errores introducidos (intencionados o no) con la adquisición de las señales de los satélites GPS. Con estos sistemas se puede llegar a errores muy pequeños, incluso menores a 1 metro en el mejor de los casos [66].

Como se ve en la tabla, el peor defecto con el uso de GPS es el tiempo de establecimiento, conocido en la jerga de GPS como TTFF (Time To First Fix). Se consideran principalmente 3 escenarios para el establecimiento del sistema, dependiendo del estado actual del receptor GPS [67]. Cuanta más información tengamos a priori, más rápido será este establecimiento. Los 3 escenarios contemplados son:

- **Frío**: No se dispone de ningún dato. Ocurre cuando el receptor está recién iniciado y no tiene estimaciones de localización válidas ni información sobre los satélites. Por tanto hay que buscar de forma sistemática todos los satélites, y al encontrar uno, esperar que envíe el "almanaque", la información básica de todos los satélites GPS, para poder escucharlos correctamente. Este almanaque se envía cada 12.5

minutos., con lo cual puede ser especialmente lento este proceso. Habitualmente se considera 15 minutos, pero en muchas ocasiones puede ser mayor. En muchas ocasiones los receptores disponen de esta información, pues al ser estática se incluye en el dispositivo o como mucho sólo hay que obtenerla la primera vez.

- **Templado**: Se dispone del almanaque y de unos datos aproximados de localización y velocidad. En este caso sólo se necesita obtener las "efemérides" de los satélites visibles (información más detallada), que se retransmite cada 30 segundos. Se suele considerar de forma práctica que este tiempo es aproximadamente 1 minuto, en el peor de los casos. Se necesita obtener la efemérides de al menos 4 satélites, con lo cual si el receptor no trabaja en paralelo se podrá multiplicar el tiempo hasta en 4.
- **Caliente**: Se disponen de datos válidos, de almanaques y efemérides. En este caso sólo hay que escuchar la posición de los satélites, algo que se realiza en un tiempo casi imperceptible. Se corresponde con el caso en el que ya estamos conectados y sólo tenemos que escuchar subsecuentes cambios de localización.

Existe una tecnología llamada *A-GPS* (GPS asistido) que permite realizar la descarga del almanaque y efemérides mediante una conexión de datos común como pueda ser 3G en el caso de una conexión móvil [68]. Esto consigue reducir el tiempo de establecimiento, muchas veces a menos de 1 minuto.

La calidad de la recepción GPS está muy condicionada por el chip receptor de GPS. Los peores no suelen realizar procesos adicionales como A-GPS o WAAS, y no tienen buena recepción de señal, además de no realizar todos los procesos de ajuste de error que se pueden. Los chips de mejor calidad, suelen realizar todos estos procesos, pudiendo llegar a precisiones de aproximadamente 1 o 2 metros y tiempos de establecimiento de menos de 1 minuto. Los chips que vienen integrados en los smartphones comerciales son habitualmente de la gama más baja, por tanto no podemos esperar que la calidad de la localización ni su tiempo de obtención sean especialmente buenos. Si necesitamos que la calidad sea mejor, siempre podremos optar por utilizar receptores GPS externos, que se pueden conectar al móvil como dispositivos, a través de la conexión USB.

Pasando a la localización por WiFi, esta se consigue en base a conocer de antemano la localización de los distintos puntos de acceso WiFi (se dispone de esta información gracias al mapeo que se ha realizado por parte de Google de las distintas redes WiFi disponibles a nivel de calle). A partir de la conexión con el punto de acceso WiFi más cercano, se puede obtener la localización del teléfono con una precisión similar al rango de efectividad del punto de acceso [69].

La localización por WiFi es, por tanto, mucho menos precisa que la localización por GPS. Sin embargo, tiene dos características que la hacen interesante: se tarda muchísimo menos en obtener las primeras señales que con GPS y el uso en interiores de la tecnología GPS es casi imposible, debido a La falta de visibilidad hacia los satélites, cosa que no ocurre al posicionar mediante WiFi. Por tanto, cuando no podemos o nos es inconveniente esperar el TTFF de GPS o cuando estamos en interiores es interesante disponer de una alternativa como WiFi, aunque sea menos precisa.

Por último, la localización mediante la red GSM es la localización menos precisa de todas, pero puede tener alguna utilidad cuando no disponemos de señal GPS o es demasiado lenta y cuando no tenemos puntos de acceso WiFi cercanos (algo bastante improbable, pero puede ocurrir) o poca batería (y por tanto desactivamos el receptor WiFi). La localización se obtiene a partir de la localización de la celda (antena) de telefonía a la cual estamos conectados dentro de la red GSM de la compañía telefónica. Esta información es fácil de obtener a partir de los datos que envía la antena, y además es muy rápida de obtener. Sin embargo, como una única celda puede dar cobertura a una zona bastante amplia puede obtenerse un error en la localización bastante considerable, más aún cuando se puede dar el caso de que no estemos conectados en ese momento a la celda más cercana (por congestión de la misma, por ejemplo).

3.3.2. Refinamiento de la localización

Como ya hemos visto, la fuente de localización más precisa es GPS, aunque todas tienen sus ventajas e inconvenientes, lo cual hace que sea interesante la posibilidad de usar todas a la vez, especialmente para paliar el problema del tiempo de establecimiento del GPS o su posible falta de visibilidad en interiores. Por tanto utilizaremos todas las fuentes a nuestra disposición para mejorar la precisión de nuestra localización y el tiempo de obtención y de refinamiento de nuestra localización [70].

Para empezar, lo más usual es obtener una localización cacheada anteriormente. Esto se puede realizar fácilmente en Android, ya que cuando cualquier aplicación accede a la localización, ésta se almacena para que cualquier aplicación pueda acceder a ella como última localización conocida. Con esta localización se almacena también su fecha, así que si es demasiado antigua se puede considerar nula y comenzar nuestro proceso de obtención de la localización directamente con la primera localización que llegue, que usualmente será de WiFi o de Cell-ID.

Por tanto, la primera localización de la que dispondremos será por lo general muy rápida (casi instantánea a ojos del usuario) pero con una precisión bastante mala. Posteriormente, con todas las fuentes de localización activadas, iremos obteniendo nuevas medidas de localización de las distintas fuentes, para realizar un proceso de refinamiento de la

localización, que vaya acercándonos cada vez más a nuestra posición correcta, disminuyendo el error en la precisión.

Principalmente hay dos factores a tener en cuenta a la hora de refinar nuestra localización: el momento de obtención de la nueva localización, y su precisión, que viene determinada, entre otros factores, por la fuente de la cual procede dicha medición, y es perfectamente consultable desde la API de Android. Si nuestro dispositivo está fijo, podremos obviar la marca de tiempo, y considerar que todas las mediciones pueden ser usadas para refinar nuestra localización. En caso contrario, tendremos que considerar el caso de que una medición más nueva pueda ser mejor que una antigua, aunque en principio tenga una precisión peor. Para ello también se suele tener en cuenta una estimación del movimiento y la velocidad, a partir de distintas mediciones de las localizaciones GPS, algo que realizan de forma automática muchos dispositivos GPS.

Utilizando estas ideas básicas de localización, fecha y precisión se pueden realizar muchos algoritmos para realizar un refinamiento progresivo de la localización. De esta manera, cuanto más tiempo estemos escuchando a la posición mejor será nuestra estimación de la misma, aunque podremos acceder a una estimación poco fiable desde el mismo primer momento. Un algoritmo básico muy simple para la realización del refinamiento se puede encontrar en la documentación de referencia de Android [70].

No obstante, para el ámbito del proyecto utilizaré los mecanismos que ofrece Google mediante los Google Play Services, que permiten realizar este refinamiento de la localización de forma más rápida y más precisa, ya que utiliza algoritmos mejores que uno tan simple como el que hemos visto anteriormente [71]. Aun así, es importante notar que el funcionamiento es esencialmente el mismo: se van obteniendo medidas de distintas fuentes con distintas precisiones y se van mezclando para ir obteniendo progresivamente una localización final cada vez más refinada y más cercana a la posición real del móvil.

4. Desarrollo del trabajo

4.1. Desarrollo del cliente

4.1.1. Introducción

Como ya se ha dicho anteriormente, para el desarrollo del cliente se ha utilizado como base la tecnología Android. Se necesitaba un dispositivo con capacidades de GPS, conexión de red, alto impacto en el mercado y sencillez de desarrollo. Todas ellas características que se cumplen en el caso de Android.

Se puede ver claramente la importancia de la aplicación para smartphone en el desarrollo, ya que será la base para la entrada de datos de todas las incidencias que vayamos a incorporar al sistema. Para que la aplicación se pueda utilizar con eficacia, se ha desarrollado para que la utilización por parte del usuario sea rápida, sencilla y eficiente.

La base de la aplicación cliente es, por tanto, la adquisición de la localización y la codificación y envío de dicha información para ser compatible con Twitter. El resto de la aplicación será una interfaz de usuario sencilla para poder realizar las operaciones necesarias de cara al usuario sin añadir más complejidad a la aplicación.

La localización se obtendrá mediante un servicio de Google Play Services, que se basa en varias fuentes distintas de localización, dando prioridad a la señal de GPS. Este servicio se encarga de gestionar las entradas de localización que van llegando, según su precisión, fuente y latencia para ir ajustando cada vez más la localización actual exacta, que se irá actualizando en tiempo real.

La información se enviará como ya hemos dicho a través de Twitter. Se utiliza la cuenta del propio usuario como emisor, para lo cual tendremos que dar permiso a la aplicación para conectarse en nuestro nombre (de forma segura, mediante la autentificación a 3 vías de OAuth). Se codifica la información de la incidencia en el propio tweet, utilizando para ello un texto en XML, con un formato que discutiremos posteriormente.

4.1.2. Actividad principal

La actividad principal de la aplicación es el punto de acceso del usuario a la aplicación, en la cual se presentan los elementos de la interfaz gráfica necesarios para su correcto uso.

De hecho, es la única actividad que será visible por el usuario, ya que la otra actividad que compone la aplicación se lanza exclusivamente para realizar la gestión de la autentificación en Twitter, y esto se realiza en segundo plano, sin necesidad de presentar información en pantalla.

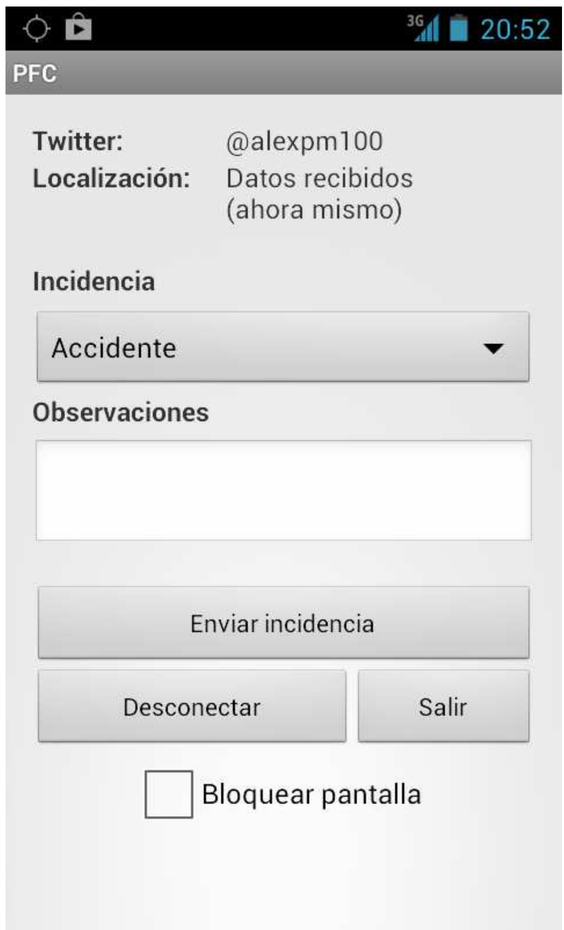


Ilustración 13 – Actividad principal del cliente Android

Como se puede ver en las capturas de pantalla, la aplicación es muy sencilla de usar. Disponemos en primer lugar de la información necesaria para que el envío de los datos se pueda realizar correctamente. Estas son las etiquetas de texto que se ven en la parte superior: Twitter y Localización.

Gracias a la primera sabremos si el sistema se ha podido conectar a nuestra cuenta de Twitter, y en caso afirmativo cual es la cuenta con la cual nos conectamos. En caso de no estar conectados a ninguna cuenta, no podremos realizar envíos de incidencias, así como si no disponemos de conexión a Internet en este momento.

La localización, como ya se ha dicho, se irá refinando conforme aumenten el número de respuestas de localización de GPS o la red. Podremos ver desde la etiqueta si se han recibido datos, se están recibiendo ahora mismo o si no se ha recibido nada. Además, podremos saber el momento de la llegada de la última actualización de la localización, ya que en caso de que sea muy lejana, podría estar desactualizada respecto a la actual, especialmente si estamos moviéndonos. El caso de no tener ninguna localización es raro, pues aunque puede ser común no recibir actualizaciones, ya que los GPS pueden tardar o incluso fallar, el servicio de localización guarda las últimas posiciones, aunque se considerarán

desactualizadas. Por último, comentar que si no se activa explícitamente la localización por GPS en las opciones, el servicio funcionara únicamente con las localizaciones por red, mucho menos precisas y fiables. Estaremos seguros de usar GPS cuando salga el icono de GPS en la barra de notificaciones.

Debajo de estos, tendremos una lista desplegable para seleccionar el tipo de incidencia a informar de entre una lista de varias incidencias y un área de texto opcional mediante la cual introducir un texto explicativo que acompañe a la incidencia.

Abajo tenemos una serie de botones cuyo funcionamiento es bastante obvio.

Mediante el botón de enviar podremos enviar la incidencia siempre y cuando se cumplan las condiciones ya explicadas (conexión a Internet, autentificación en Twitter y localización obtenida).

EL botón de conectar/desconectar se utiliza para autentificarnos en Twitter mediante OAuth o, en caso contrario, desactivar las credenciales de la cuenta actual. El proceso se detallará más adelante.

Podremos salir mediante el botón de Salir o la tecla Atrás de Android.

Finalmente, tenemos un Check Box llamado “Bloquear pantalla”, que permitirá dejar la aplicación activa (sin entrar en suspensión). Esto permite un uso más sencillo, ya que si queremos avisar de una incidencia rápidamente, no tendremos que pasar por el proceso de desbloqueo del teléfono.

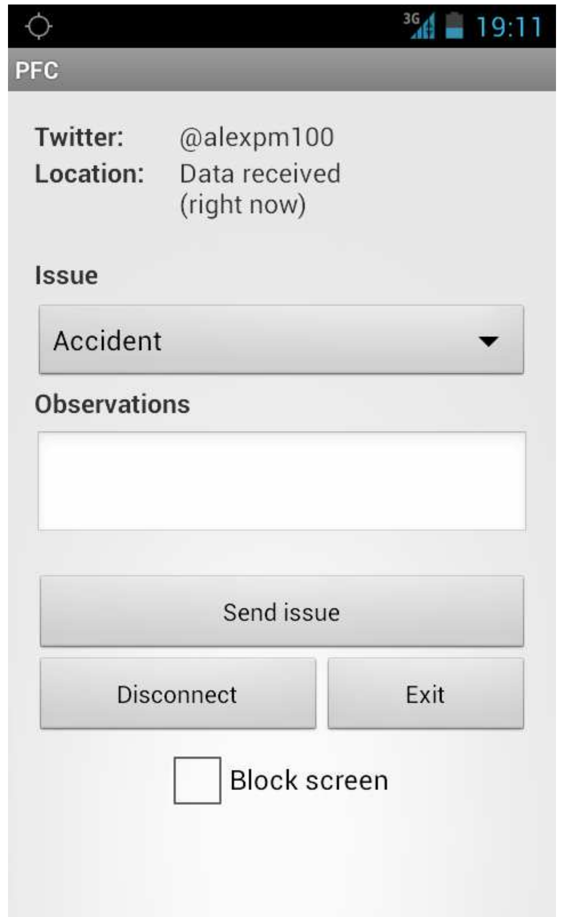


Ilustración 14 – Actividad principal en inglés

La aplicación se ha desarrollado de manera que se pueda realizar la internacionalización de manera sencilla. Siguiendo el esquema de Android para el tratamiento de cadenas de texto externalizadas (el recurso strings.xml), ha sido sencillo dejar la aplicación adaptable a la internacionalización. Como prueba y demostración de que el sistema funciona correctamente, se ha realizado la aplicación tanto en español, como en inglés.

Ilustración 15 – Actividad principal en modo landscape

Como se puede ver en las pantallas, la aplicación cliente también se adapta a la orientación de la pantalla. De esta manera, se puede usar la aplicación en vertical (modo normal) o en horizontal (modo landscape).

4.1.3. Servicio de localización

El servicio de localización es probablemente la parte más importante del cliente, ya que es el encargado de asegurarse que la localización que obtenemos es lo más precisa posible, y se obtiene además en tiempo real.

Por ello, para la implementación final he decido utilizar uno de los servicios de Google, denominado *Location Services* del paquete de servicios generales para Android, *Google Play Services*.

Los servicios de Google Play, son una serie de servicios, disponibles para todas las aplicaciones del dispositivo. Esto se hace así por dos razones: evitar la instalación repetida de librerías que pueden estar usándose en varias aplicaciones y permitir el uso común de recursos y la compartición de información entre aplicaciones. El único punto a criticar de estos servicios es que obligan a que la aplicación tenga como objetivo mínimo dispositivos con Android 2.2. Sin embargo, ya Google considera obsoletas las versiones anteriores, y de hecho, ninguno de los nuevos terminales que se venden tiene versiones anteriores a la 2.2, así que de hecho, llegaríamos prácticamente al 100% del mercado Android haciendo objetivo a partir de la versión 2.2.

Algunos de los servicios que ofrece Google Play Services son:

- Mediación para publicidad (Google Ads o AdMob)
- Paso de información entre aplicaciones
- Gestión de la autentificación (OAuth, entre otros)

- Obtención de datos de Google Play (mercado de aplicaciones)
- Gestión de juegos de Google Play (usuarios, compras, logros, multijugador, etc.)
- Gestión de la localización
- Comunicación con Google Maps
- Servicios de cobro (Google Wallet)
- Conexión con la red social Google Plus

Evidentemente, lo que más nos interesa es la obtención de la localización. La gran ventaja que tiene esta forma respecto a obtener la localización de forma manual (directamente de GPS) son varias.

La primera es la simplicidad. La cantidad de operaciones que se deben realizar para mantener la localización refinada a partir de las múltiples llegadas de localización (location fix) se simplifican, ya que el propio paquete se encarga de dicha gestión, así no tendremos que preocuparnos de los tiempos y precisiones de las distintas fuentes. De esta manera, podremos obtener en cualquier momento la mejor precisión posible, sin importar como se consiga, así como escuchar a nuevas localizaciones. Por tanto, en vez de conseguir los nuevos "fix" y tener que tratarlos, se devolverá el nuevo refinamiento, y por tanto, conforme pase el tiempo obtendremos un mejor aproximamiento a la localización real.

La segunda y más importante tiene que ver con la posibilidad del servicio de estar ejecutándose en segundo plano y compartido con otras aplicaciones. En el caso de que otras aplicaciones anteriormente o simultáneamente al cliente del proyecto estén accediendo a los recursos de localización estos ya estarán obteniendo datos y mejorando la localización. Por tanto, aprovecharemos los esfuerzos de otras aplicaciones, en vez de competir por los escasos recursos del GPS. Así mejoraremos la calidad del sistema a nivel global y empezaremos desde mucho antes a obtener coordenadas fiables para el envío de las incidencias. Con respecto a esto, también cabe destacar que si otra aplicación ha inicializado la conexión GPS, ya estará inicializada. Esto es algo muy importante, ya que en los peores casos puede llegar a tardar incluso minutos, en los cuales sólo dispondríamos de la localización por red, bastante menos precisa que la de GPS.

Otra de las ventajas es la gestión automática de las fuentes de localización. En vez de tener que gestionar a mano cuales son los distintos proveedores de localización y si están activos y tenemos permisos para ellos, el servicio sabe directamente cuales se pueden usar y hace uso de todos ellos, sin importar cuales son.

Por último, hay que comentar que este método de obtener la localización soluciona un fallo que aparece en algunos dispositivos. En estos

dispositivos el uso del GPS está unido a Google Maps, lo cual hace que si no se ejecuta la aplicación de Google Maps (da igual) no se obtenga acceso al GPS, o que no funcione correctamente. En esos casos se obligaba a tener una instancia de Google Maps sin uso, con el correspondiente gasto de recursos que ello supone en el smartphone, además de la problemática para el desarrollo.

4.1.4. Autentificación en Twitter

La autentificación en Twitter requiere de hacer uso de los tokens de acceso OAuth, que son los usados por Twitter para la autentificación de usuarios. Como ya hemos visto, esto requiere la autentificación a tres vías entre el proveedor (Twitter), el usuario y la aplicación.

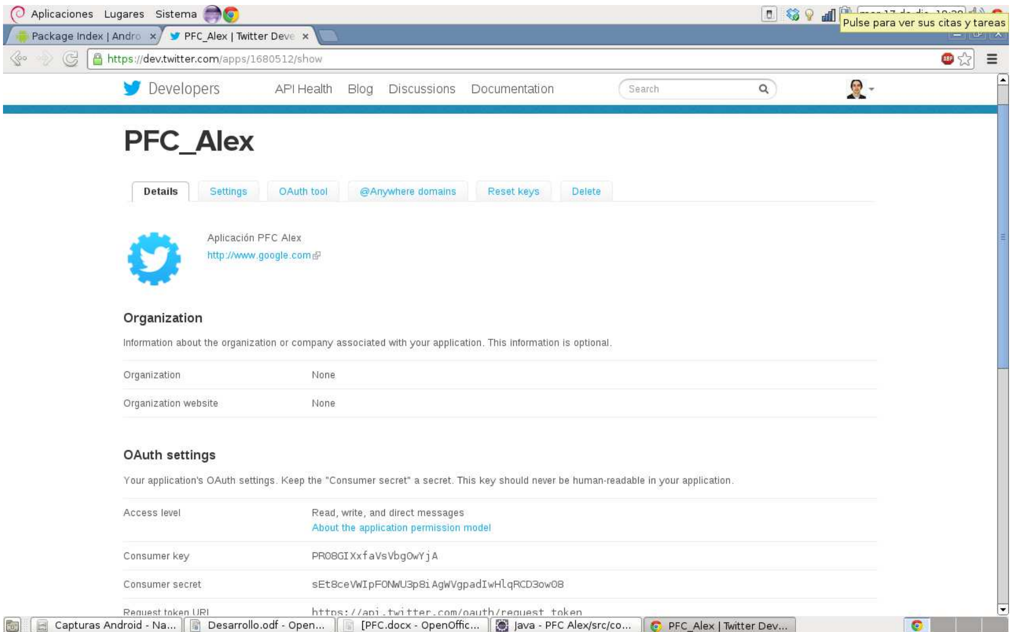
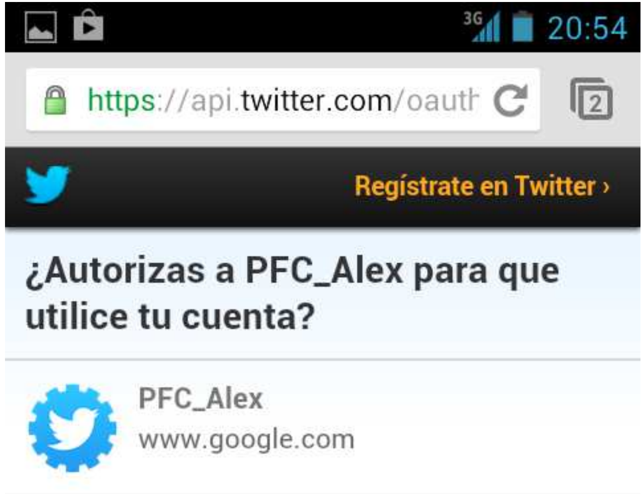


Ilustración 16 – Página de desarrolladores de Twitter

Para ello hemos de dar primero de alta la aplicación en Twitter y seleccionar los permisos necesarios. En mi caso, ya que se necesita enviar y recibir tweets seleccionaré los de lectura y escritura. Con esto obtendremos los códigos OAuth que necesitamos. El par Access token/Access token secret se utiliza para firmar peticiones con nuestra propia cuenta. Estos los usaremos para la obtención de los nuevos tweets de incidencias desde el servidor. Para el cliente será más complicado, ya que necesitaremos generar un Access Token distinto para cada usuario gracias a la autentificación a 3 vías.

A partir de nuestro par de claves Consumer key/Consumer secret podremos enviar al usuario a la página de obtención de la clave (request URL). En esta página Twitter le pedirá usuario y contraseña al usuario que se va a identificar (esto sólo lo verá Twitter y nosotros no podremos obtener su contraseña). Cuando Twitter valide al usuario devolverá el

token de acceso (Access token/Access token secret) a nosotros mediante la dirección de callback.



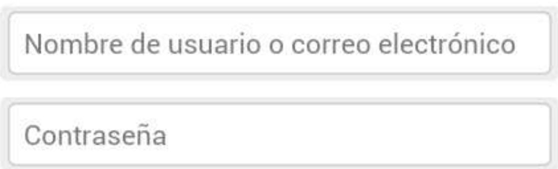


Ilustración 17 – Autentificación OAuth en Twitter

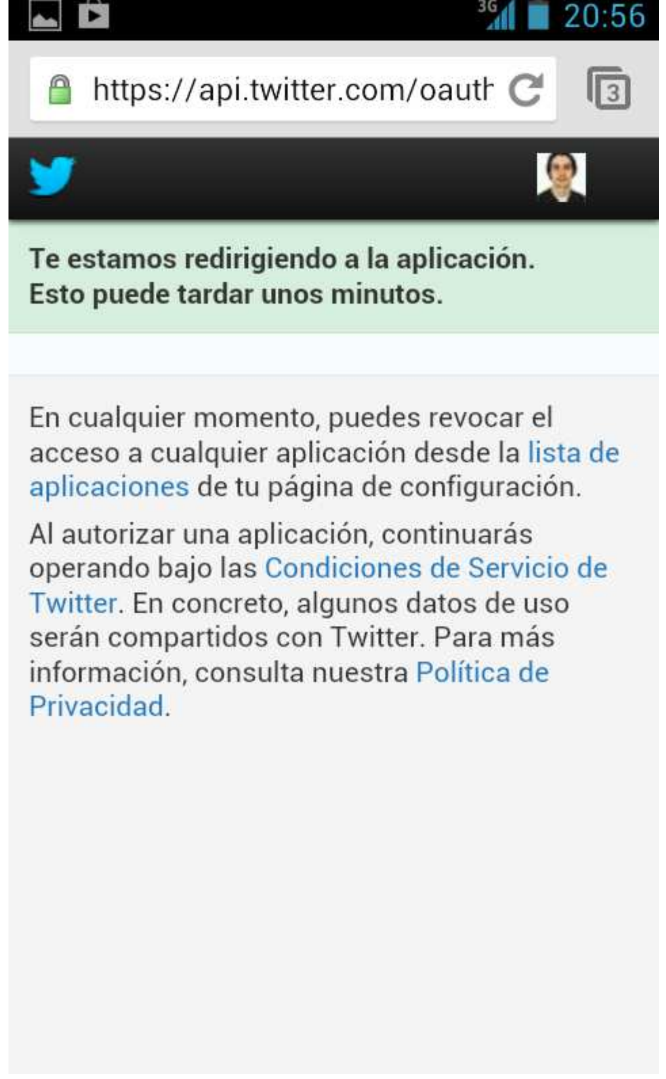


Ilustración 18 – Respuesta de la petición OAuth de Twitter

Para poder realizar el proceso de comunicación web mediante el móvil he utilizado una llamada al navegador por defecto para el envío de información y un servidor Apache para la recepción de la clave tras la autentificación. Esto se realiza mediante las librerías de *Apache Commons*, que traen la implementación básica de un servidor Apache para su uso en Java (además de otras utilidades) [72].

Todo este proceso se gestionará desde una actividad Android propia, ya que es compleja y consume mucho tiempo. De esta manera lo separaremos del bucle principal y lo ejecutaremos en segundo plano.

Esta actividad se encarga de lanzar el navegador y el servidor que recoge el callback de Twitter con el token de acceso del usuario. Esto se realiza de forma asíncrona mediante una AsyncTask, ya que requiere de conexión a Internet. Una vez completada la recepción se pasa a guardar el token y el secreto en la configuración de la aplicación (*SharedConfiguration*), que es

compartida entre las distintas actividades de la aplicación, y se da por finalizada la actividad, retomando el control la actividad principal.

Una vez obtenido el token y de vuelta en la actividad principal, hay que obtener el nombre de usuario Twitter, operación que servirá también para validar el token. Para ello haremos uso de la API de Twitter, a la cual tenemos pleno acceso una vez obtenido el token. En concreto utilizaremos la librería Twitter4J, que encapsula las llamadas de servicios Web de Twitter a una interfaz de Java basada en clases. Esto simplifica enormemente el trabajo, al no tener que encargarnos de parsear los mensajes de petición y respuesta de la API de Twitter [73]. Esta será la librería que usaré para el uso de Twitter en el resto del proyecto.

4.1.5. Envío de datos

Como ya se ha dicho, el envío de las incidencias se va a realizar a través de la plataforma de Twitter. Se utilizará como emisor la cuenta del usuario autentificada en el paso de autentificación, que queda determinada de forma única por el token de acceso obtenido y almacenado en la aplicación. Como receptor se utilizará una cuenta fija del servidor, que será la que siempre reciba las incidencias de todos los usuarios.

En la incidencia se envían dos datos principales: el usuario emisor, que se enviará directamente sin que hagamos nada gracias a los metadatos del tweet; y los datos de incidencia, que deben ser codificados como una cadena, ya que es lo único que se permite enviar como contenido en un tweet.

Además, hay que tener en cuenta la limitación de longitud de Twitter, que solo permite 140 caracteres, con lo cual hay que limitar el tamaño del texto de observación, ya que es el único de longitud variable, que podría hacer que nos pasemos de la longitud máxima del tweet, en cuyo caso no se nos permitiría enviar el tweet correctamente.

El formato escogido para la codificación de las incidencias como cadena de texto es XML. Para la codificación se ha utilizado la librería JDOM. Esta librería permite la manipulación sencilla de documentos DOM, mediante llamadas simples, y está escrita enteramente en Java, a diferencia de antecesoras similares, con lo cual es bastante eficiente y se integra mejor en el sistema [74].

Ilustración 19 – Captura de ejemplo de un tweet codificando una incidencia

Pasaremos ahora a ver el formato de los tweets y su XML. Nos basaremos en este ejemplo y analizaremos sus elementos:

```
@alexpm100 <inc x="36.8472785" y="-2.4672288" t="1384022187872" i="7" text="obras" />
```

- **Cuenta de usuario Twitter de la aplicación**: Se debe empezar con la cuenta de usuario de la aplicación para que dicha cuenta lo reciba como mención y pueda recopilar las distintas incidencias.
- **Etiqueta inc**: La incidencia se codificará en una etiqueta XML autoconclusiva (sin hijos) denominada inc.
- **Atributo x**: La latitud de la coordenada.
- **Atributo y:** La longitud de la coordenada.
- **Atributo t**: Marca de tiempo del mensaje.
- **Atributo i**: Código entero que define el tipo de incidencia.
- **Atributo text**: El texto explicativo que acompaña a la incidencia. Es opcional.

Una vez construida la cadena de texto completa, se envía a la cuenta de la aplicación a través de la librería Twitter4J, como ya habíamos dicho anteriormente. Como esto requiere de conexión a Internet, por un lado deshabilitaremos la opción en caso de no haber conexión, y por otro lado, el proceso se realizará mediante una tarea asíncrona (AsyncTask). Durante el proceso se mostrará un diálogo de espera y al terminar se mostrará un mensaje de éxito/fracaso. El éxito será el caso más normal, el fracaso ocurrirá sólo si sobrepasamos el límite de uso de la API de Twitter (en cuyo caso tendremos que esperar 15 minutos hasta el siguiente intento), si el usuario ha revocado manualmente su token de acceso, en cuyo caso tendrá que volver a autentificarse mediante OAuth (presionando el botón

de Conectar) o en el caso de una repentina perdida de conexión a Internet.

Ilustración 20 – Incidencia en proceso de envío

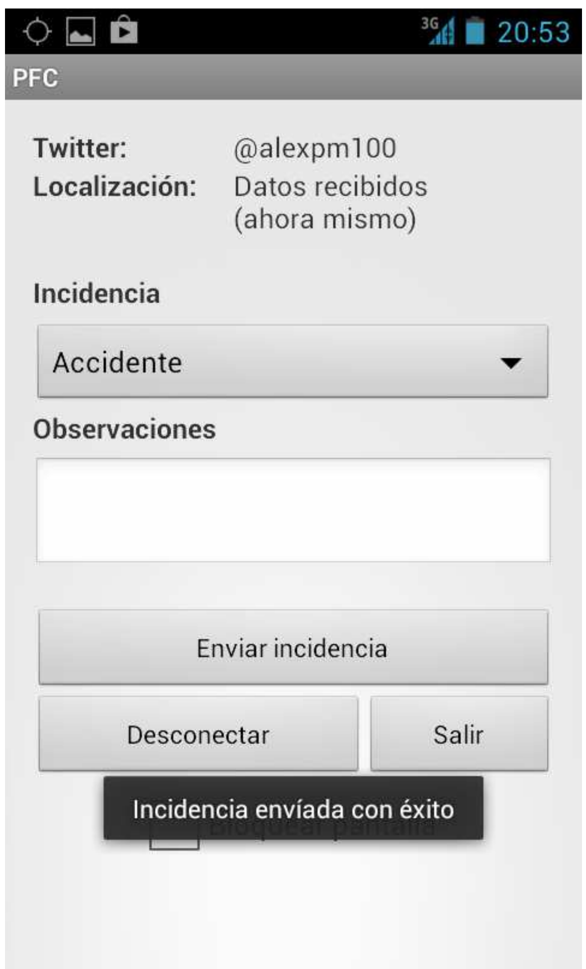


Ilustración 21 – Incidencia enviada con éxito

4.2. Desarrollo del servidor

4.2.1. Introducción

Para el servidor se ha desarrollado una aplicación Web que permite monitorizar en tiempo real todas las incidencias enviadas por los usuarios de la aplicación móvil. Para ello se monitorizan todos los tweets recibidos como mención por la cuenta de Twitter de la aplicación y se procesarán las incidencias que contienen, incluyéndose en una base de datos interna a la aplicación.

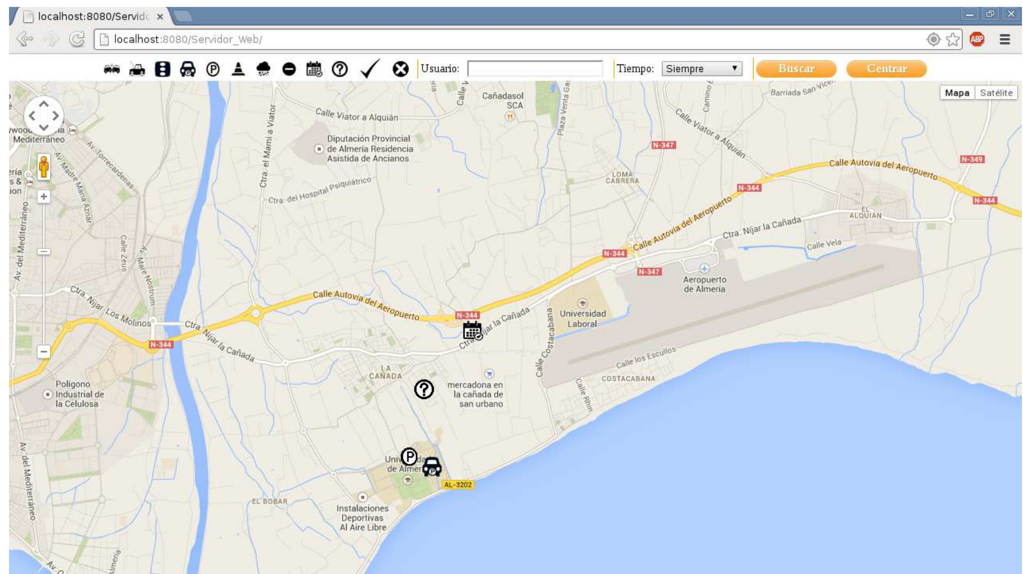


Ilustración 22 – Ejemplo de la aplicación web

A cada tipo de incidencia se le asignará entonces un icono diferente, que se posicionará en su coordenada correspondiente en un mapa de Google Maps. Podremos observar el resto de la información relevante (usuario, fecha y hora, observaciones) pasando el ratón por encima de cualquier marcador en el mapa.

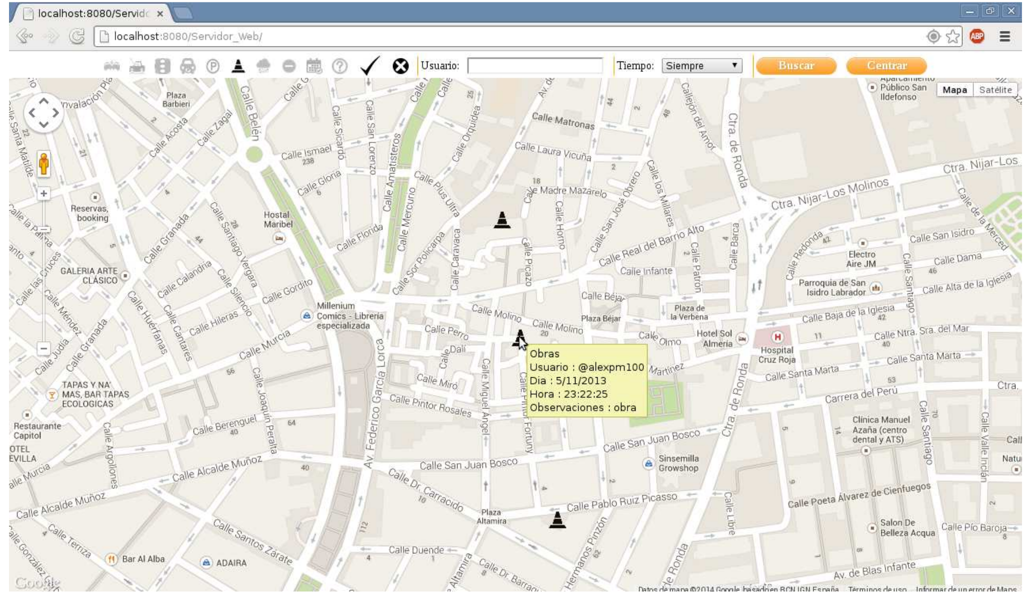


Ilustración 23 – Información detallada de una incidencia

Además, disponemos de una barra de búsqueda en la parte superior, que permite filtrar las búsquedas por distintos criterios. De esta manera será más sencillo usar el mapa, evitando muchos marcadores que no nos interesan. En concreto, como se ve en la captura, se puede filtrar por tipo de incidencia (todas, una, o varias), usuario (opcional) y rango de fechas (entre varios a elegir). Una vez establecidos los parámetros de búsqueda podremos actualizar el mapa para que sólo se muestren las incidencias

que cumplen los criterios, haciendo click en el botón de buscar. A partir de ese momento las nuevas incidencias recibidas que cumplan los criterios se añadirán directamente al mapa sin necesidad de refrescar la página.

Finalmente, se puede ver que el mapa es completamente manipulable, sin ninguna restricción de posición o zoom, para que el usuario pueda navegar libremente por él. En caso de querer ver de forma óptima todos los marcadores a la vez se puede hacer click en el botón Centrar, que fijará de forma automática el centro y el nivel de zoom.

4.2.2. Arquitectura

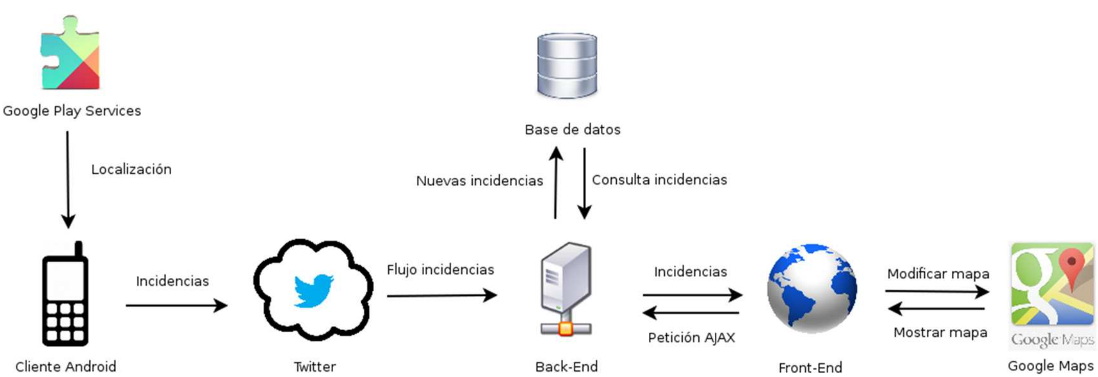


Ilustración 24 – Esquema básica de la arquitectura del sistema

En el diagrama se puede observar la arquitectura básica, sin entrar en muchos detalles, del sistema desarrollado. Me he centrado en la parte del servidor, ya que es la más compleja y la que más interesa analizar en este punto.

Como ya he comentado en apartados anteriores, a partir del cliente instalado en el smartphone y haciendo uso del servicio de localización GPS de Google Play Services, obtenemos la localización y rellenamos la incidencia, que se envía a Twitter como un tweet dirigido a la cuenta de la aplicación.

En este momento entra en juego la parte servidora de la aplicación. Dicho servidor se divide en dos partes distintas aunque interrelacionadas. Estas son el back-end y el front-end.

El back-end es la parte interna del servidor, aquella que hace el trabajo pesado y no es vista por los usuarios finales. Su misión fundamental es la de conectarse a la cuenta de Twitter para recoger los nuevos tweets y traducirlos en incidencias y encargarse de la conexión a la capa de datos, encargada de gestionar la inserción de nuevas incidencias y de las consultas de incidencias que se requieran por parte de la interfaz Web. Para esto se ha utilizado la tecnología Java, en concreto JSP (JavaServer Pages), para el renderizado de páginas web dinámicas (de forma similar a otras tecnologías del lado de servidor, como PHP) [75]. Esto se ha hecho para poder realizar la capa de comunicación con Twitter una única vez,

basándome en la misma librería que he utilizado para la aplicación Android, la librería Twitter4J.

Por otro lado tenemos el front-end. Éste es el encargado de la parte final de cara al usuario, con la cual se comunica directamente con la aplicación. Por tanto la responsabilidad del front-end es la de gestionar toda la interfaz de usuario y la comunicación con otras partes (el back-end y Google Maps). Todo esto se ejecuta por tanto directamente en el cliente, a través de un navegador Web o browser (Google Chrome, Mozilla Firefox, Internet Explorer, etc.) Por tanto, se deben utilizar las tecnologías web típicas del lado del cliente, como son HTML (HyperText Markup Language) para la estructura de la página web [76] , CSS (Cascading Style Sheets) para el diseño [77] y JavaScript para la lógica y elementos dinámicos [78], además de para la comunicación con la API de Google Maps, que está escrita en JavaScript.

Además de eso hay que tener en cuenta que una vez renderizada la página web por el servidor se encuentra completamente en el navegador del cliente, y por tanto incapaz de comunicarse con el servidor. Por ello no podríamos obtener las incidencias entrantes a no ser que refrescásemos completamente la página, además de tener que generarla de nuevo (con el consiguiente aumento de la carga de trabajo para el servidor). Por ello se ha utilizado AJAX para implementar la actualización de la información del front-end, realizando consultas dinámicas al back-end sin requerir una carga total de la página web.

4.2.3. Obtención de información

La obtención de incidencias tiene 2 pasos bien diferenciados: la actualización inicial una vez que arranca el servidor y el modo de escucha que va recibiendo nuevas incidencias de Twitter.

La actualización inicial consiste en hacer una petición a Twitter de todas las menciones a la cuenta de la aplicación con fecha posterior a la de la última incidencia registrada en el sistema. Se intentan procesar los tweets recibidos por esta consulta según el formato XML que deben cumplir las incidencias recibidas por la aplicación. Para ello se hace uso de la librería JDOM, de forma similar (pero inversa) al envío de incidencias desde el cliente. Aquellos tweets que cumplen la especificación son procesados y añadidos al sistema, y el resto son desechados, ya que son mensajes de Twitter comunes, y no incidencias. Estas incidencias pasarán a la capa de persistencia y serán guardados en la base de datos.

Para obtener las nuevas incidencias que van llegando una vez que el servidor ya ha sido iniciado, la solución más sencilla sería realizar nuevas peticiones a Twitter de forma periódica, similares a la actualización inicial. No obstante, es una mala solución, ya que sería una sobrecarga de trabajo, al realizar multitud de peticiones cuando no ha llegado ningún tweet potencial de ser incidencia. Además, haría saltar en unas pocas

peticiones el límite de Twitter, haciendo que nuestro servidor quedase sin conexión efectiva a Twitter para captar nuevas incidencias hasta pasado el límite de Twitter (15 minutos).

La solución correcta es utilizar la parte de la API de Twitter llamada Streaming API, ya introducida anteriormente. Gracias a esta API podemos suscribirnos desde nuestra aplicación a un stream de usuario que nos permitirá recibir en tiempo real los tweets entrantes. Cada uno de estos tweets será analizado en busca de incidencias con su formato XML correcto.

En la librería Twitter4J la llegada de los tweets en flujo se modela mediante eventos. Se registra un listener, en el cual se implementa un método por cada una de las posibles acciones asociadas al stream del usuario suscriptor. En concreto nos interesa *onStatus*, que se activa por cada tweet nuevo recibido en la cuenta de la aplicación. Llegados a este punto, el procesamiento es similar al del caso inicial, se comprueba si sigue el formato XML de incidencias, y en caso afirmativo se extraen los datos y se guardan en el sistema, mediante la capa de persistencia.

4.2.4. Persistencia

Dado que obtener y procesar los datos de incidencias completamente desde cero por la interfaz de Twitter puede ser increíblemente lento (especialmente si ya hay muchas incidencias enviadas) y además hará saltar rápidamente las limitaciones en número de peticiones de la API de Twitter, será necesario guardar los datos de las incidencias ya obtenidas mediante Twitter de forma persistente, en una base de datos.

Como ventaja añadida, esto también permite que, en el caso de tener algún problema con Twitter (página inactiva, problemas de limitaciones o credenciales, etc.) el sistema seguiría funcionando con los datos de la base de datos local, y lo único a lo que no tendríamos acceso momentáneamente sería a la inserción de nuevas incidencias.

Para la implementación de la base de datos se ha utilizado MySQL [79] como SGBD y PHPMyAdmin [80] sobre Apache (con el paquete de software XAMPP/LAMPP [81]) para la administración de dicho sistema. Para la interfaz entre el código y la BD, se han utilizado los drivers oficiales JDBC de MySQL, que permiten realizar consultas de forma sencilla a una base de datos MySQL a través de cualquier aplicación Java. La elección de esta base de datos viene dada por su simplicidad de uso, junto con su potencia a la hora de afrontar diseños con alta carga de consulta (pocas inserciones con respecto a muchas búsquedas).

El diseño de la base de datos es muy simple, con únicamente dos tablas: una con los datos de las incidencias, debidamente indexados por los campos de consulta; y otra con los nombres de usuario, para evitar llamadas de más a la API de Twitter, que sólo permite operar con ids de usuario. Es importante destacar que no se guarda ninguna información

sensible sobre las cuentas de usuario, como contraseñas o datos personales, únicamente se dispone del id de Twitter y el nombre de usuario (ambas son información pública). Para información completa del diseño de la base de datos, se puede consultar el apartado correspondiente de los anexos.

Las operaciones básicas que soporta la capa de persistencia de la aplicación son:

• **Inserción de nuevas incidencias**: Una vez parseada una incidencia nueva desde Twitter, se registra en el sistema insertándola en la BD.

• **Inserción de nuevos usuarios**: Esto ocurre en el caso de que una nueva incidencia sea enviada desde una cuenta de Twitter no registrada en la BD.

• **Consultas parametrizadas**: En función de los parámetros de búsqueda especificados por el usuario desde el front-end, se devolverán las incidencias que se muestran finalmente en el mapa, filtradas por dichos parámetros.

4.2.5. Diseño de la interfaz web

Como ya se ha dicho, el desarrollo de la interfaz Web se ha realizado con las tecnologías comunes de front-end: HTML, CSS, JavaScript y AJAX.

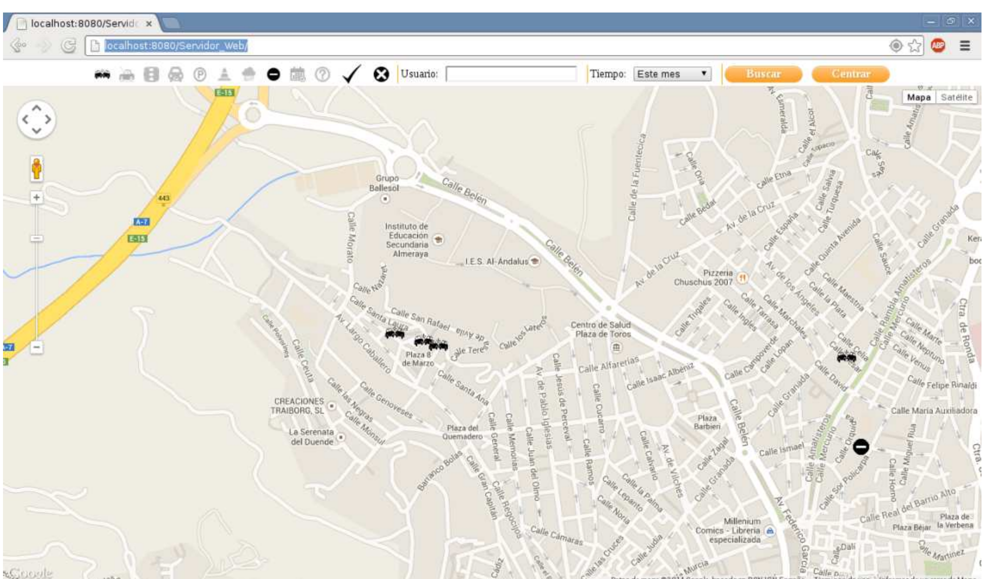


Ilustración 25 – Captura de ejemplo de la aplicación Web

La página principal, que se puede ver en la captura de pantalla, se divide en dos componentes principales: la barra de búsqueda y el mapa. Además, se encarga de inicializar todos los elementos necesarios para la ejecución de la página: recursos de JavaScript (propios y la API de Google Maps) y las hojas de estilos CSS. La maquetación de los elementos se ha realizado mediante el uso de capas o *divs*, posicionados con el uso de CSS.

El selector de tipos de incidencias se ha realizado con JSP. Iterando sobre los distintos tipos de incidencias registrados en el sistema conseguimos no tener que modificar la página principal en el caso de modificar los tipos de incidencias activos del sistema. El resto de componentes de la barra de búsqueda son componentes estándar de entrada de usuario de HTML. Estos son:

- **Enlaces**: Todos los botones son enlaces a un código JavaScript estilizado como un botón mediante CSS. Esto incluye los iconos de tipos de incidencia para filtrar y los botones de Buscar y Centrar.
- **Input**: Cuadro de texto para la entrada del usuario Twitter por el cual filtrar las incidencias.
- **Select**: Lista de selección para elegir entre distintos rangos temporales para filtrar las incidencias.

Cabe destacar que los elementos de interfaz no forman parte de un formulario típico en HTML, sino que se encuentran sueltos en la página. De esta manera procesamos dicha entrada con JavaScript y posteriormente se envía al servidor mediante AJAX (sólo si se requiere carga de información desde el servidor). Además, como ya he comentado, evitaremos recarga completa de la página con cada cambio en la barra de búsqueda.

La capa de Google Maps es únicamente una capa vacía, ya que actúa únicamente como Viewport para la API de Google Maps. Esto se configura en la propia página y se inicializa con la inicialización de la API de Google Maps.

El código JavaScript asociado a la página se encarga de inicializar las variables de la página y gestionar el estado de las diferentes partes de la aplicación. También se encarga de la gestión de Google Maps a través de su API y de las peticiones AJAX hacia el servidor, las cuales comentaré en sus respectivos apartados.

4.2.6. API de Google Maps

Como ya se ha comentado varias veces, la API de Google Maps está escrita en JavaScript y se ejecuta directamente desde el navegador del cliente. Al inicializar esta API se obtiene en la página Web un mapa de Google Maps completamente funcional, el cual puede ser modificado mediante llamadas a funciones de la API. Estas llamadas se han implementado en un archivo JavaScript denominado gmaps.js

Para la inicialización se necesita especificar las opciones básicas del mapa, que son el nivel inicial de zoom (se puede cambiar más adelante) y el tipo de mapa, que se ha configurado como ROADMAP (el tipo por defecto que se utiliza en un mapa tradicional de Google Maps). También se podrá cambiar por el usuario a la vista de satélite, desde el propio mapa.

En la inicialización también es necesario especificar la capa de la web que se va a utilizar como viewport para el mapa. En caso contrario no se posicionará el mapa dentro de la página. El mapa ocupará todo el espacio reservado para la capa en CSS, que deberá estar vacía para no interferir con el renderizado del mapa en la web.

El trabajo más importante que voy a realizar con respecto al mapa consiste en añadir marcadores al mapa para posicionar las distintas incidencias en éste. Para ello es necesario crear un objeto de tipo *Marker*. Este objeto se posiciona mediante sus coordenadas (con el objeto *LatLng*, que especifica latitud y longitud) y se le ofrecen una serie de opciones. En mi caso le añado un icono personalizado, dependiendo del tipo de incidencia; un tamaño para el icono, dependiendo de la forma del icono; y un texto explicativo (que se mostrará al pasar el ratón por encima del marcador) que incluye: nombre, usuario, fecha, hora y observaciones.

Para añadir un marcador al mapa será necesario hacer uso de la función *setMap(map*) en dicho marcador. Para eliminarlo se utilizará *setMap(null).* También habrá que añadir/eliminar la coordenada correspondiente de un objeto de tipo *LatLngBounds*, que especifica los límites que rodean de forma mínima a todos los marcadores posicionados. Esto se usa en la función de centrar, que llama a la función de API *fitBounds(bounds).* Esta función centra el mapa y modifica el nivel de zoom para que todos los marcadores entren en la zona de visión de la forma más ampliada posible. La función se ejecuta de forma asíncrona, con un efecto visual para el desplazamiento. Esto hace que a veces pueda tardar un poco en terminar de ejecutarse, en especial cuando hay muchos marcadores posicionados. Sin embargo, podremos seguir trabajando sin bloquear el mapa, al ser una función que se ejecuta de forma concurrente a otras llamadas de la API.

Para resumir el proceso, tenemos que realizar las siguientes operaciones en Google Maps para posicionar las incidencias entrantes:

- Si las incidencias vienen de una nueva búsqueda (y no del stream) se borra el mapa (todos los marcadores y los límites).
- Se genera el marcador con JavaScript a partir de los datos recibidos de la incidencia.
- Se añade el marcador al mapa con *marker.setMap*
- Se añade el marcador a los límites con *bounds.extend*

4.2.7. AJAX

Vamos a ver ahora la implementación de la carga asíncrona de datos en la web con AJAX. Esto se ha usado para poder cargar nuevas incidencias al front-end de forma asíncrona, sin tener que recargar la página completa. Además, gracias a AJAX podremos monitorizar la llegada de nuevas incidencias e incluirlas a nuestro mapa en tiempo real.

Las llamadas se gestionan desde un archivo JavaScript, ajax.js. En este archivo se implementa en primer lugar la infraestructura para poder realizar las llamadas AJAX.

Para empezar se necesita de un objeto *XMLHTTPRequest*. Esto es una interfaz, compatible con la mayoría de navegadores web, que permite hacer desde el navegador peticiones HTTP de forma independiente a la propia petición para el renderizado de la página. Esta interfaz utiliza XML para comunicarse con el servidor y permite recibir respuestas y procesarlas en el cliente (con JavaScript) de forma asíncrona.

Para realizar una petición necesitamos:

- **La URL de destino**: En mi caso se utiliza una página JSP en el servidor que gestiona la llamada a la base de datos para obtener las incidencias. En la URL debemos generar los parámetros que se pasan al servidor (los parámetros de búsqueda) en forma de cadena de texto. Esto se realiza en JavaScript al construir la cadena de búsqueda.

- **Generar la petición**: La petición se envía como una petición HTTP GET de forma normal. En este punto podríamos enviar algunos datos extra con la petición, algo que no es necesario, pues todos los datos necesarios se encuentran como parámetros codificados en la URL

- **Callback**: Se requiere de una función de callback, que es la que se llamará una vez se haya respondido la petición por parte del servidor. Esta función JavaScript procesará los datos devueltos por el servidor.

Una vez realizado este proceso llegaremos a la función de callback, en mi caso *getOutput*. Aquí podemos obtener los datos que necesitemos de los devueltos por el servidor y procesarlos. La respuesta viene codificada en XML, aunque se puede acceder directamente a una serie de parámetros. En concreto yo he necesitado usar dos:

- **readyState**: Es un valor numérico que especifica el estado de la petición. El estado 4 significa completado. Cualquier otro estado no nos interesa, ya que significa que ha ocurrido algún error, y en ese caso no debemos procesar los datos.

- **responseText**: Es el texto de respuesta, como texto plano. En este caso lo he usado en vez de *responseXML*, ya que la respuesta es sencilla y usar XML incrementaría de forma innecesaria el overhead de la comunicación AJAX.

Si la respuesta es correcta (cuando readyState == 4) el texto generado como respuesta por el servidor mediante su página JSP será un texto en formato CSV con las incidencias separadas por saltos de línea y los atributos de cada incidencia separados por punto y coma. En caso de que alguno de los textos incluyera alguno de esos caracteres, se escaparán de forma debida para evitar problemas, convirtiéndolos respectivamente en

[LINEFEED] y [SEMICOLON], los cuales se desescaparán a la hora de generar el texto de cara al usuario.

Cada una de las incidencias, una vez leída de esta respuesta CSV se procesará y añadirá al mapa tal como se ha explicado en el apartado de Google Maps.

Finalmente, tras la llegada de la petición AJAX tendremos los datos hasta el momento. Ahora falta poder ir recibiendo los datos nuevos que lleguen desde el stream consumido por el back-end. Para ello se usara una técnica conocida como *polling*. El polling consiste en realizar de forma periódica una petición al servidor, en espera de que se obtengan datos nuevos. La llamada de AJAX que se utilizará en el polling será similar a la de la primera carga, con la excepción de que se utiliza un parámetro extra, denominado *lastTime*, que incluye la marca de tiempo de la última incidencia recibida por el front-end, para evitar que se repitan los datos iniciales, con el consecuente aumento de la carga de trabajo y el tiempo de computación y de acceso a datos.

Es importante aclarar que los datos recibidos con polling son incrementales, y se incorporan al mapa sin sustituir a los anteriores, ya que son actualizaciones provenientes en tiempo real del stream de tweets de Twitter. En el caso de la primera carga de AJAX se omite el parámetro lastTime y se cargan todas las incidencias que cumplen el filtrado. Estas incidencias si son sustitutivas de las anteriores, y por tanto se debe limpiar el mapa antes de su inserción.

Como último punto, una vez que se cambian los parámetros de búsqueda por parte del usuario, se podrán recargar las incidencias desde cero, al hacer click en buscar. En este caso se cancelará el polling activado actualmente y se lanzará una nueva petición AJAX con los nuevos criterios de búsqueda. Esta petición se considera nueva petición, y por tanto debe eliminar todas las incidencias posicionadas en el mapa y cargar todas las incidencias que cumplan los criterios. Finalmente, tras recibir dichas incidencias, se volverá a restablecer el polling con los nuevos criterios de búsqueda y así volveremos a estar en estado de escucha esperando la llegada de nuevas incidencias del stream.

5. Conclusiones finales

5.1. Objetivos alcanzados

El objetivo principal del proyecto era realizar una versión funcional del sistema propuesto, objetivo que se ha alcanzado de forma correcta.

Se ha implementado completamente, por tanto, un sistema de información geográfica que permite, mediante los datos de localización de un dispositivo móvil, geolocalizar incidencias de tráfico en un mapa, y visualizarlas de forma correcta y fácilmente accesible mediante un servidor web que se encarga de presentar visualmente la información a los usuarios finales.

En este desarrollo se ha conseguido un diseño robusto y sencillo de usar, con las opciones de usabilidad necesarias para que el sistema sea lo más sencillo posible, pero con la potencia suficiente para realizar todas las operaciones necesarias.

Las dos partes principales (cliente y servidor) que componen el sistema realizan todas sus funcionalidades de forma correcta y rápida, y están preparados para funcionar de una forma desacoplada, de tal manera que se podría cambiar fácilmente cualquiera de ellos sin ningún problema.

Por tanto, se puede decir que el desarrollo cumple todas las expectativas y resuelve los problemas que se planteaban a la hora de realizar el proyecto.

5.2. Objetivos formativos alcanzados

Para el desarrollo del proyecto he requerido del estudio y del uso de una serie de técnicas y herramientas indispensables para la implementación del sistema de incidencias.

Con respecto al cliente, he tenido que estudiar los aspectos claves de la tecnología Android, para poder implementarlo en smartphones. Además, he tenido que profundizar en aspectos con especial relevancia para el proyecto, como la obtención de la localización, con énfasis en la tecnología GPS junto con sus ventajas e inconvenientes. También es especialmente importante la comunicación por red, haciendo uso de las tecnologías de servicios web, especialmente REST para la comunicación con Twitter, aunque se han estudiado las principales tecnologías que se utilizan en la actualidad, como XML, también importante en la implementación final.

Para la parte del servidor se han utilizado también tecnologías de servicios web para la comunicación con Twitter y con Google Maps. Además, se ha requerido del uso de tecnologías web variadas para el desarrollo del sistema final. Algunas de estas tecnologías serían HTML, JavaScript, JSP, XML o AJAX. Ha sido especialmente importante el estudio de la comunicación entre back-end y front-end mediante AJAX y el mantenimiento de la persistencia, haciendo uso de técnicas de cacheo y de bases de datos.

Para las distintas partes de la aplicación se ha seguido un diseño basado en componentes, haciendo uso de APIs siempre que ha sido posible y encapsulando el código de manera que fuese lo más reutilizable posible (especialmente teniendo en cuenta las partes comunes entre cliente y servidor).

Finalmente, se ha realizado un análisis detallado del sistema completo, haciendo uso de UML para detallar los componentes más importantes del sistema.

5.3. Extensiones futuras del proyecto

El proyecto realizado se presta a muchas posibilidades de extensión, ya que el campo de la geolocalización ofrece un abanico bastante grande de posibilidades.

Ya que el sistema se ha diseñado para ser lo más flexible y sencillo posible, se podría adaptar a cualquier otro campo de aplicación diferente al de las incidencias de tráfico. Se podría aprovechar todo el sistema de obtención de la localización con GPS bajo un cliente Android, el sistema de mensajes en XML sobre Twitter y la obtención de dichos mensajes de forma continua mediante el servidor. Cambiando únicamente el concepto de incidencia por cualquier otro concepto basado en la posición, podríamos hacer cualquier aplicación geográfica, en la cual podríamos localizar nuestra posición fácilmente.

Además, se podrían añadir elementos más complejos, tales como recintos cerrados o rutas, formados por varios puntos geográficos.

Con respecto al caso concreto de las incidencias de tráfico, una opción muy interesante sería la implementación de un sistema de reconocimiento de voz, que permita el envío de incidencias mientras se conduce, sin necesidad de tener que hacer uso de las manos para realizar el envío de la incidencia. Esto permitiría que un conductor hiciese uso de la aplicación mientras conduce, de forma similar a un manos-libres.

6. Bibliografía

[1] http://queue.acm.org/detail.cfm?id=1142044.

[2] http://docs.oracle.com/javase/tutorial/rmi/overview.html.

[3] C. A. M. Machueca, «Estado del Arte: Servicios Web».

[4] http://www.opengroup.org/soa/source-book/soa/soa.htm#soa_definition.

[5] http://www.stargroup.uwaterloo.ca/~mhamdaqa/publications/Cloud_Computing_Uncovered.pdf.

[6] http://www.ws-i.org/deliverables/matrix.aspx.

[7] http://tools.ietf.org/html/rfc2616.

[8] http://tools.ietf.org/html/rfc821.

[9] http://tools.ietf.org/html/rfc959.

[10] https://http2.github.io/http2-spec/.

[11] https://tools.ietf.org/html/rfc2818.

[12] http://www.w3.org/TR/REC-xml/.

[13] http://www.w3.org/MarkUp/SGML/.

[14] http://www.w3c.es/Divulgacion/GuiasBreves/XHTML.

[15] http://java.boot.by/wsd-guide/ch01s02.html.

[16] http://www.techmynd.com/advantages-disadvantages-of-xml/.

[17] http://www.json.org/.

[18] http://www.json.org/js.html.

[19] http://www.w3.org/TR/soap/.

[20] http://www.tutorialspoint.com/soap/soap_header.htm.

[21] http://www.tutorialspoint.com/soap/soap_body.htm.

[22] http://schemas.xmlsoap.org/soap/encoding/.

[23] http://www.tutorialspoint.com/soap/soap_encoding.htm.

[24] http://www.tutorialspoint.com/soap/soap_fault.htm.

[25] R. N. T. Roy T. Fielding, «Principled Design of the Modern Web Architecture».

[26] http://www.sei.cmu.edu/reports/02tn015.pdf.

[27] http://www.w3.org/TR/wsdl20-primer/.

[28] http://www.tutorialspoint.com/wsdl/wsdl_elements.htm.

[29] https://helloreverb.com/developers/swagger.

[30] http://jsondoc.org/.

[31] http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/.

[32] http://www.w3.org/Submission/wadl/.

[33] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec.

[34] http://www.innoq.com/blog/st/2010/03/uddi_rip.html.

[35] http://labs.safelayer.com/es/articulos/integracion-de-servicios-de-seguridad/228-esbspi.

[36] https://twitter.com/.

[37] http://www.internetadn.es/blog/conoce-pic-twitter-com.

[38] https://dev.twitter.com/docs/api.

[39] https://dev.twitter.com/docs/streaming-apis.

[40] http://oauth.net/.

[41] https://dev.twitter.com/docs/auth/oauth.

[42] http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/.

[43] https://dev.twitter.com/docs/api/1.1.

[44] https://dev.twitter.com/docs/platform-objects.

[45] https://dev.twitter.com/docs/rate-limiting/1.1.

[46] https://dev.twitter.com/docs/rate-limiting/1.1/limits.

[47] https://code.google.com/apis/ajax/playground/.

[48] http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/.

[49] https://code.google.com/apis/console/.

[50] https://developers.google.com/maps/documentation/webservices/?hl=es.

[51] http://developer.android.com/google/index.html.

[52] https://developers.google.com/translate/v2/getting_started.

[53] https://developers.google.com/maps/documentation/business/?hl=es.

[54] https://developers.google.com/maps/documentation/?hl=es.

[55] maps.google.com.

[56] https://developers.google.com/maps/documentation/javascript/reference?hl=es.

[57] https://developers.google.com/maps/documentation/android/map.

[58] http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal.

[59] http://www.openhandsetalliance.com/press_110507.html.

[60] http://reload4btech.blogspot.com.es/2012/11/comparisons-of-all-android-versions.html.

[61] http://www.xataka.com/moviles/android-4-4-kitkat-pocas-novedades-a-simple-vista-pero-con-muchos-cambios-para-el-futuro.

[62] http://developer.android.com/guide/index.html.

[63] http://www.colorado.edu/geography/gcraft/notes/gps/gps_f.html.

[64] https://web.archive.org/web/20120329111058/http://ngs.woc.noaa.gov/FGCS/info/sans_SA/docs/GPS_SA_Event_QAs.pdf.

[65] http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/library/documents/media/waas/2892bC2a.pdf.

[66] http://www.puertos.es/ayudas_navegacion/sistemas_navegacion_por_satelite/Sistemas_GPS_y_DGPS.html.

[67] http://www.gpsinformation.net/main/gpslock.htm.

[68] http://www.wpcentral.com/gps-vs-agps-quick-tutorial.

[69] http://www.zdnet.com/blog/networking/how-google-and-everyone-else-gets-wi-

fi-location-data/1664.

[70] http://developer.android.com/guide/topics/location/strategies.html.

[71] http://developer.android.com/google/play-services/location.html.

[72] http://commons.apache.org/index.html.

[73] http://twitter4j.org/en/index.jsp.

[74] http://www.jdom.org/mission/index.html.

[75] http://www.oracle.com/technetwork/java/overview-138580.html.

[76] http://www.w3.org/TR/1999/REC-html401-19991224/.

[77] http://tools.ietf.org/html/rfc2318.

[78] https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript?redirectlocale=en-US&redirectslug=JavaScript%2FAbout_JavaScript.

[79] http://www.mysql.com/.

[80] http://www.phpmyadmin.net/home_page/index.php.

[81] https://www.apachefriends.org/es/index.html.

[82] https://www.apachefriends.org/download.html#4043.

[83] http://tomcat.apache.org/download-70.cgi.

7. Anexos

7.1. Manual de uso

7.1.1. Instalación del cliente

Lo primero que se necesita para instalar la aplicación es disponer del paquete *apk*, que contiene toda la información necesaria para que un móvil Android pueda ejecutar la aplicación. Para ello será necesario compilar la aplicación y exportarla al formato apk, para posteriormente subirla al móvil y poder instalarla.

Esto se puede hacer fácilmente con el IDE Eclipse y las herramientas de desarrollo de Android (ADK). Para ello se selecciona la opción de Exportar, y posteriormente se selecciona la opción de Android.

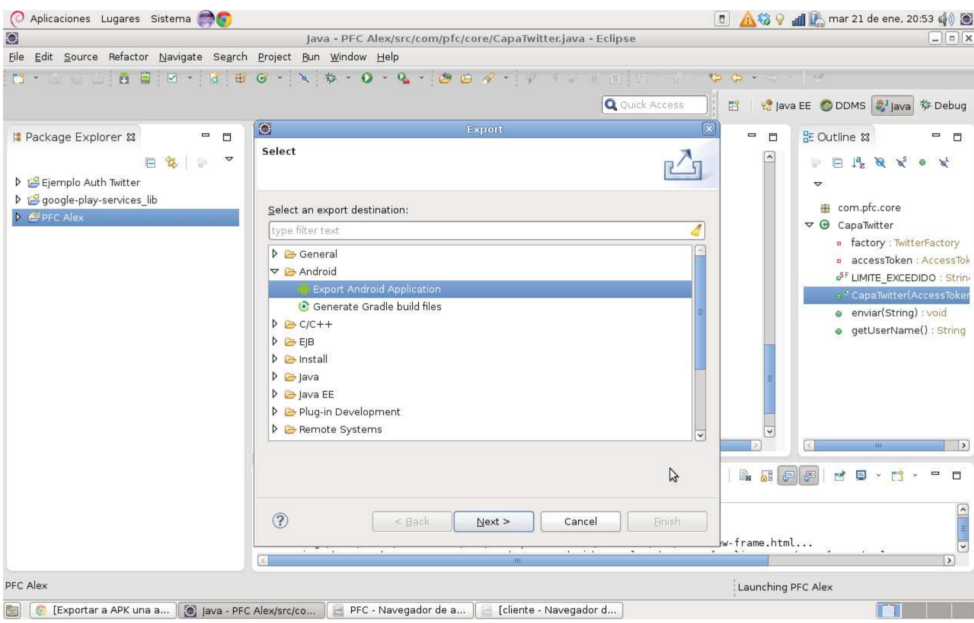


Ilustración 26 – Exportación de la aplicación Android

Después será necesario firmar la aplicación con una clave pública criptográfica. Para ello tendremos que crear una *keystore* e introducir una nueva clave de desarrollo. Esto se hacce introduciendo una serie de datos de referencia como el nombre del desarrollador, localización o caducidad de la clave. Además, se fijará una contraseña que bloqueará el acceso a dicho clave si se carece de la contraseña.

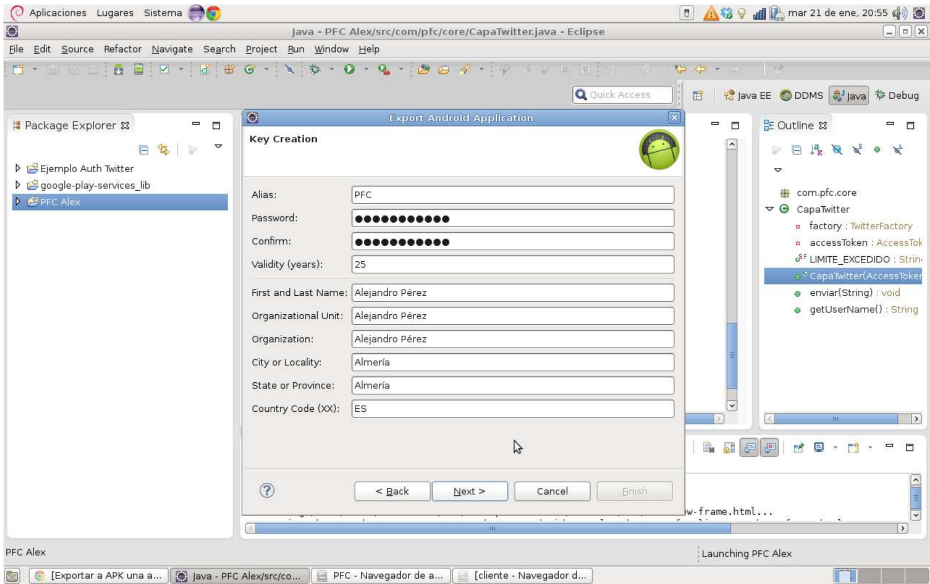


Ilustración 27 – Creación de una clave de desarrollo Android

Tras terminar de exportar la aplicación, dispondremos de un archivo apk que contiene comprimidos todos los archivos necesarios del proyecto.

Para instalar la aplicación tendremos que subir este archivo al teléfono por cualquier medio (transferencia por Internet, cable USB, tarjeta SD, etc.)

Una vez tengamos el archivo en el teléfono deberemos instalar el paquete. Para ello lo más cómodo es hacer uso de un gestor de archivos que permita la instalación sencilla de estos paquetes. Personalmente, mi elección ha sido *File Expert*, pero se puede utilizar cualquier gestor.

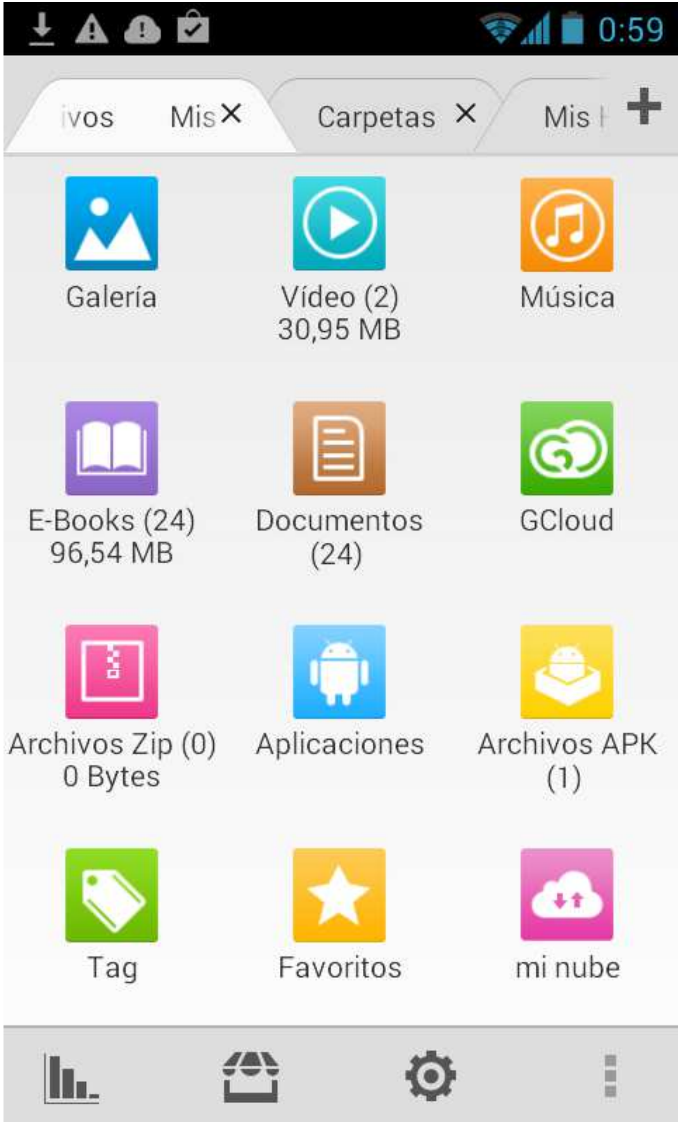


Ilustración 28 – Captura del gestor de archivos con el APK.

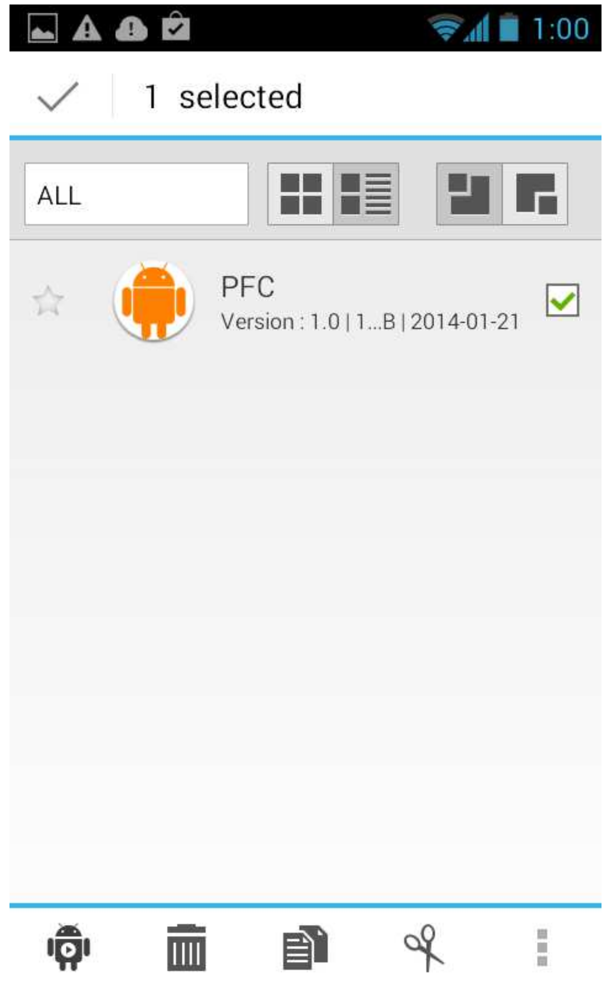


Ilustración 29 – Selección del APK.

Al seleccionar el APK de la aplicación se instalará haciendo uso del instalador de paquetes de Android. Al activarse el instalador se nos requerirá permitir a la aplicación el acceso a la conexión a Internet y a la localización, que son los permisos necesarios para que la aplicación se ejecute correctamente.

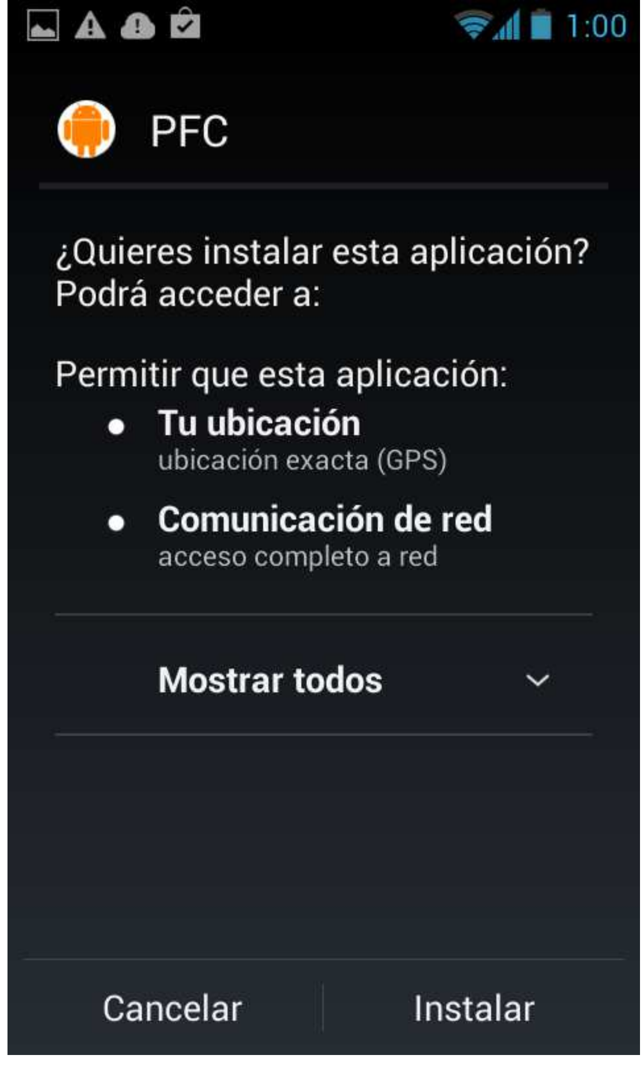


Ilustración 30 – Permisos de la aplicación Android

Una vez aceptados los permisos, la aplicación se instalará y ya podremos usarla dentro del teléfono.

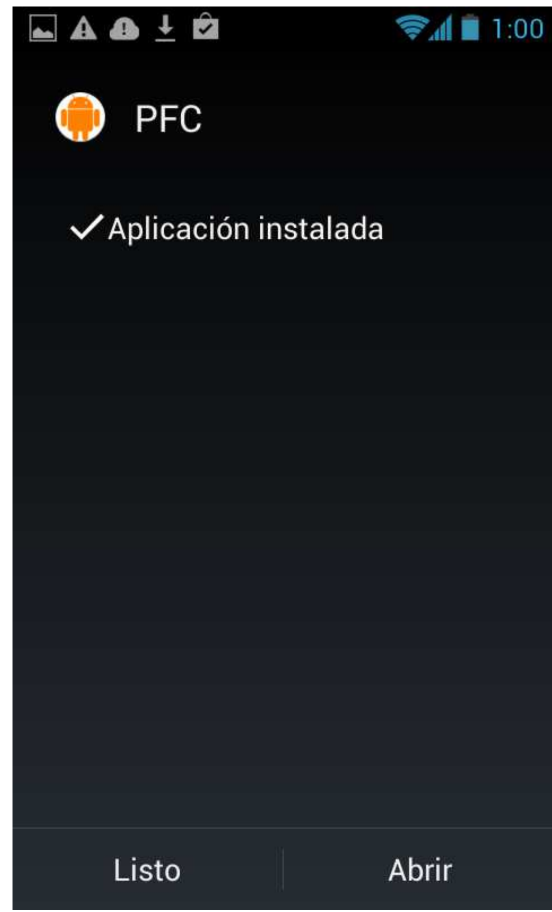


Ilustración 31 – Fin de la instalación

7.1.2. Uso del cliente

Una vez iniciada la aplicación, se accederá directamente a la actividad principal, que gestiona la interacción básica con el usuario.

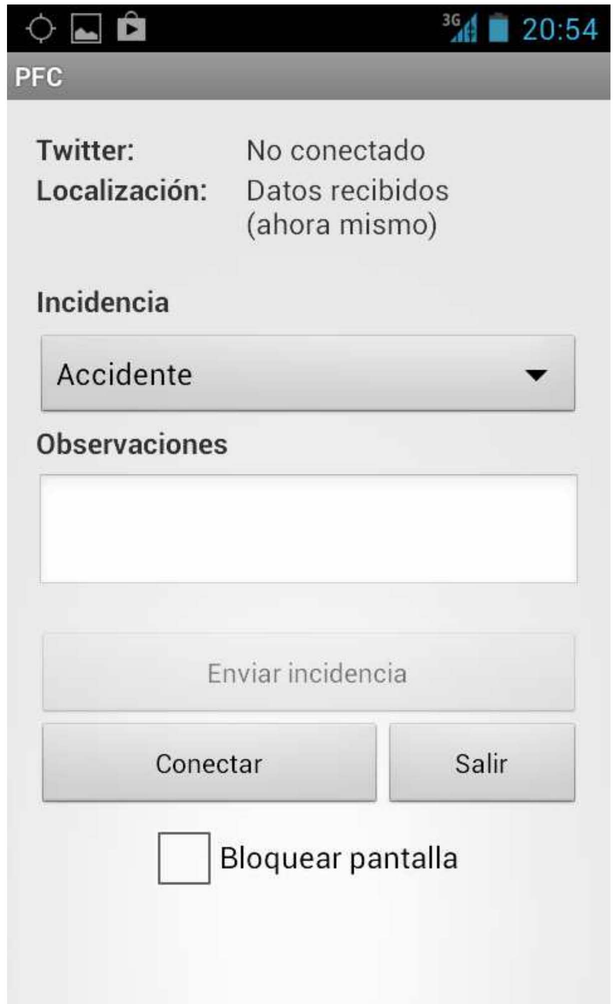


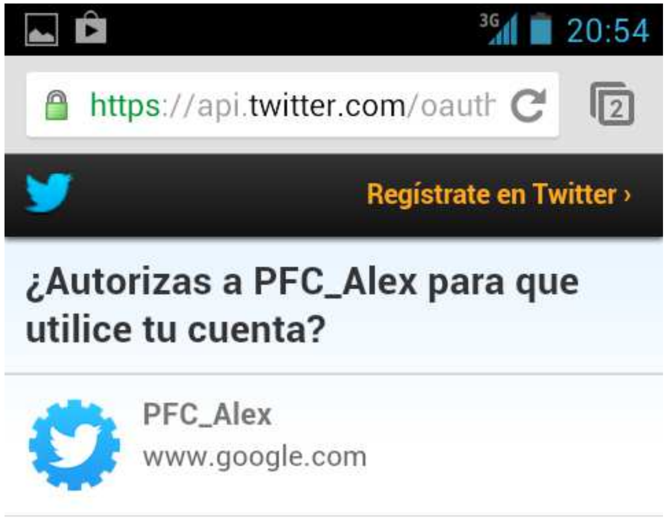
Ilustración 32 – Actividad principal

Lo primero que se observa son los datos de estado de la parte superior. Lo primero es la cuenta de Twitter del usuario. En caso de no haber realizado la conexión de la cuenta (mediante el proceso de autentificación) saldrá "No conectado", y no podremos enviar incidencias.

Lo siguiente es la localización. Generalmente las primeras localizaciones llegan casi instantáneamente, pero puede ocurrir en alguna ocasión que se retrasen un poco en llegar. Por tanto en este campo se especifica al usuario si ya se dispone de datos de localización o no. Si no se dispone de ellos no se podrá enviar la incidencia. También se especifica entre paréntesis cuanto tiempo hace que llegó la localización, para que el usuario decida si usar una localización especialmente lejana en el tiempo o esperarse a tener una más nueva.

Debajo tenemos un selector del tipo de incidencia, en el cual decidiremos el tipo de incidencia a informar de una lista desplegable de incidencias. Además, podremos acompañar la incidencia con un texto descriptivo. Este texto es opcional, aunque recomendable si queremos especificar alguna información adicional.

Para conectarnos a nuestra cuenta de Twitter pulsaremos el botón de conectar, que comenzará el proceso de autentificación en Twitter. Esto hará que se redireccione al usuario a la página de autentificación de Twitter, en la cual se le pedirá que confirme que quiere dar permiso a nuestra aplicación, para lo cual tendrá que introducir su usuario y contraseña.



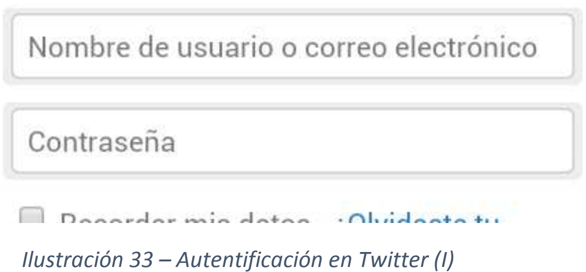


Ilustración 33 – Autentificación en Twitter (I)

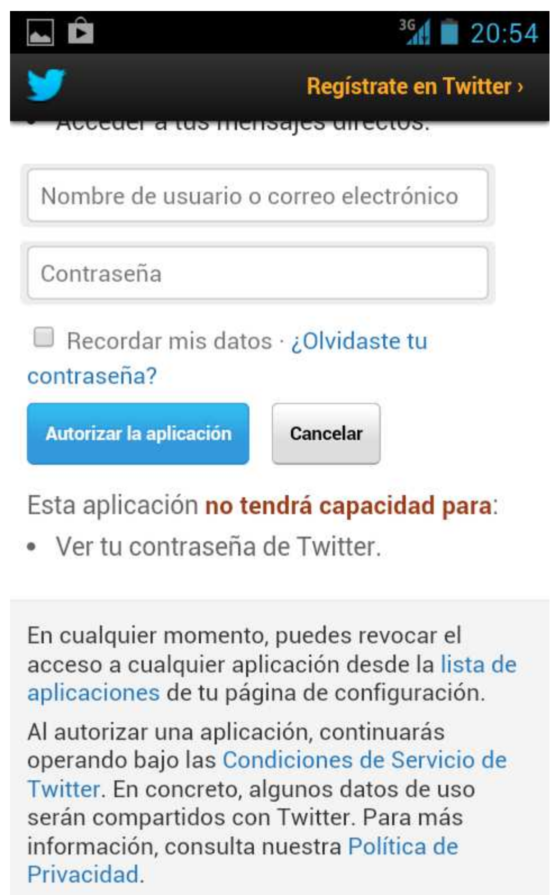


Ilustración 34 – Autentificación en Twitter (II)

Una vez completado el proceso se nos redireccionará de vuelta a la aplicación, y ya tendremos acceso a la cuenta de Twitter a la que hayamos dado acceso.

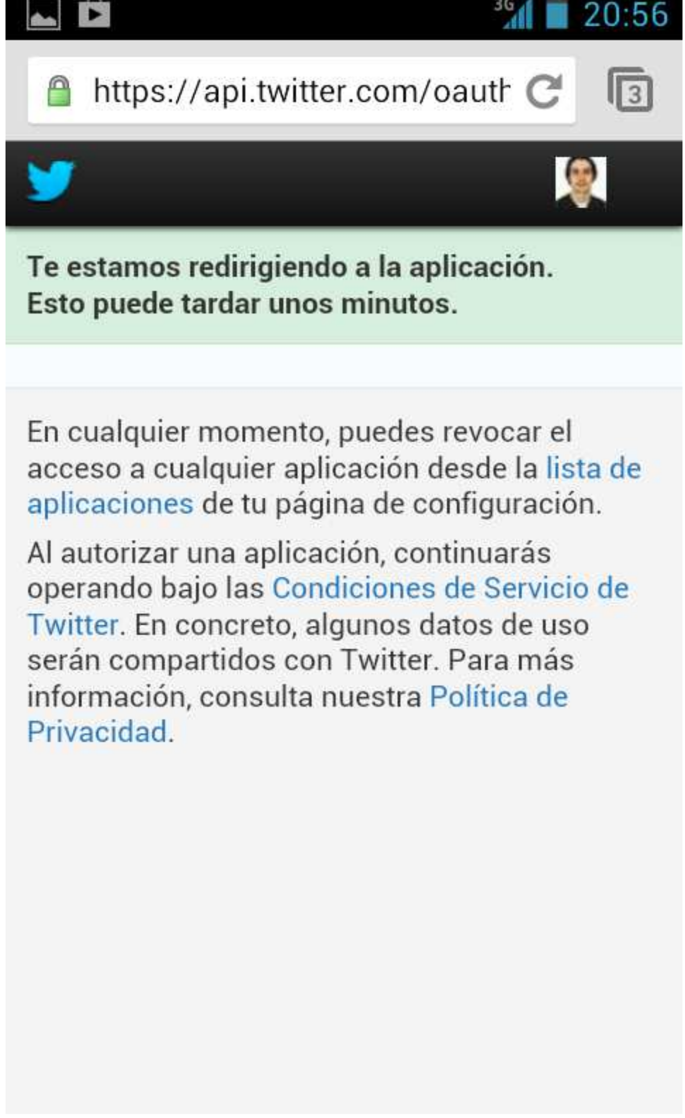


Ilustración 35 – Redirección a la aplicación

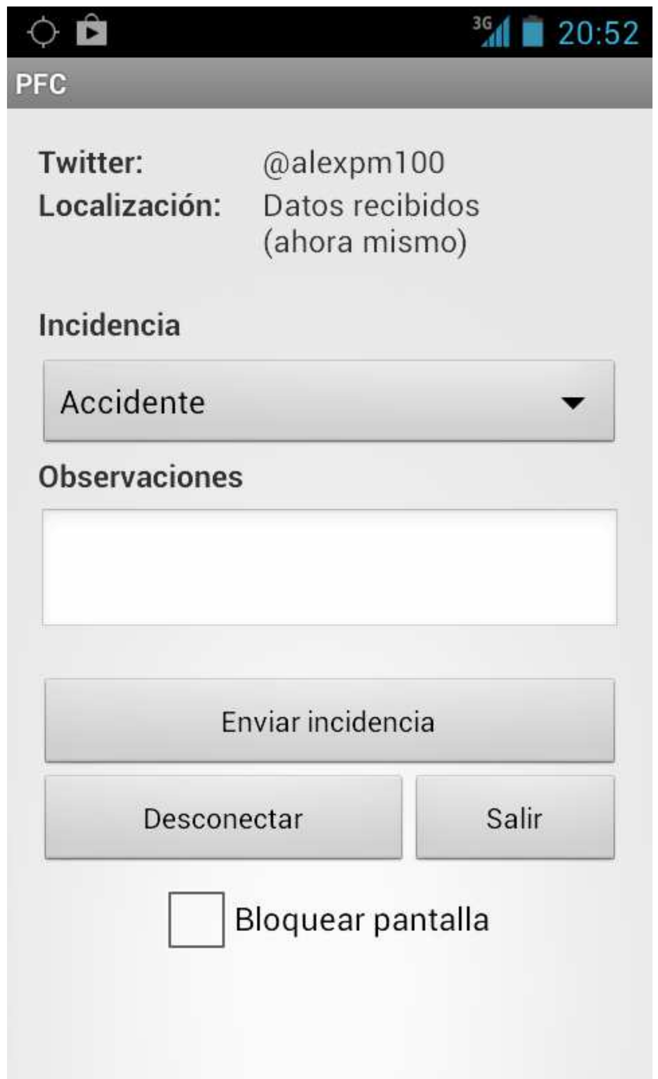


Ilustración 36 – Aplicación Android con Twitter activo

Una vez obtenidas las credenciales tendremos acceso indefinido a la cuenta de Twitter desde la aplicación incluso aunque la cerremos o apaguemos el teléfono, ya que se guardan internamente los datos de acceso. Si el usuario desea de usar la cuenta de Twitter o quiere cambiar a otra nueva podrá hacer uso del botón Desconectar, que eliminará los tokens de acceso a la cuenta de la memoria del teléfono. Después podrá volver a conectar si lo desea, repitiendo el proceso descrito anteriormente.

Para enviar la incidencia sólo hay que pulsar el botón de “Enviar incidencia”. Se enviarán los datos de localización más actualizados, haciendo uso de la cuenta de Twitter del usuario y de los datos introducidos por el mismo desde la interfaz de la aplicación.

Finalmente, el checkbox de “Bloquear pantalla” sirve para evitar que el teléfono entre en suspensión. Esto es de utilidad si queremos tener la aplicación fácilmente disponible durante un buen tiempo.

Para salir podremos pulsar el botón de “Salir” o presionar la tecla “Atrás” del teléfono.

Una última nota importante sobre la localización. El teléfono intentará acceder a todas las fuentes de localización que el usuario tenga activas. Por tanto para disponer de una mejor localización se recomienda que se activen las redes WiFi (incluso aunque usemos conexión de datos) y la conexión GPS.

7.1.3. Instalación del servidor

Para la instalación del sistema del servidor será necesario instalar una serie de programas servidores en la máquina que haga de servidor.

El primero de estos programas será XAMPP, que se utiliza para acceder a la base de datos MySQL y a una interfaz web para su gestión, phpMyAdmin. Para ello entramos en la web de XAMPP [82] y descargamos el paquete. Posteriormente, lo descomprimiremos en alguna carpeta en el servidor.

Ilustración 37 – Descarga de XAMPP

Para ejecutar XAMPP habrá que ejecutar el siguiente comando en la ruta de instalación:

```
lampp start
```

Para parar la ejecución se escribirá:

```
lampp stop
```

El siguiente programa que necesitaremos para ejecutar el sistema será un servidor Apache Tomcat, que será el servidor que genera las páginas webs a partir de Java, además de encargarse de la gestión de la base de datos y

la entrada de información. Podemos descargarlo desde la página oficial de Apache [83].

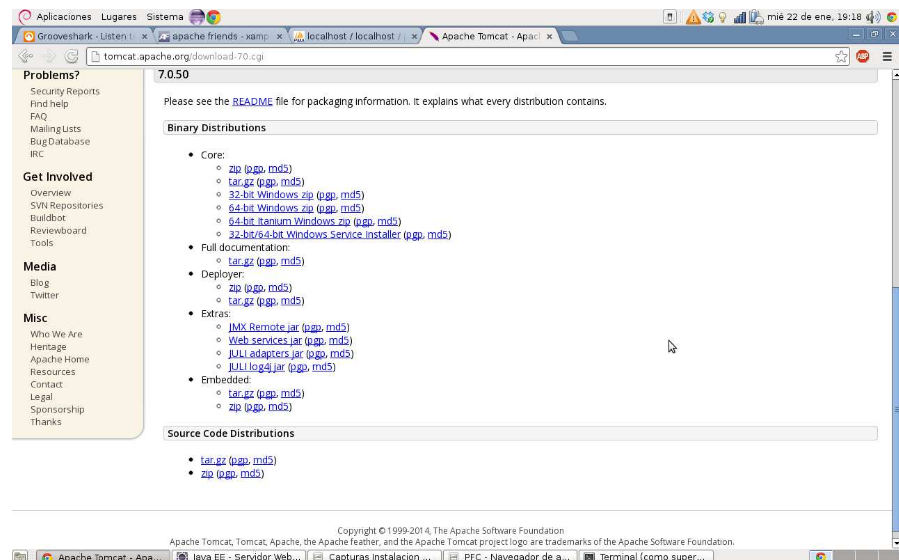


Ilustración 38 – Descarga de Apache Tomcat

Para instalarlo se descomprime en la carpeta que se desee. Después habrá que hacer una serie de ajustes para poder utilizarlo correctamente. Estos son los siguientes:

- **Establecer variables de entorno**
  - CATALINA_HOME : La ruta de instalación de Tomcat
  - JAVA_HOME : La ruta de la instalación de Java
  - JRE_HOME : La ruta de instalación de Java
- **Añadir permisos a la carpeta**: Se añadirán permisos de ejecución a los ejecutables de la instalación con el comando:

```
chmod +x *.sh
```

Después de realizar la instalación podremos usar los siguientes comandos para iniciarlo y pararlo, respectivamente:

```
sh startup.sh

sh shutdown.sh
```

Una vez hecho esto podremos acceder al servidor a través de la URL:

```
localhost:8080
```

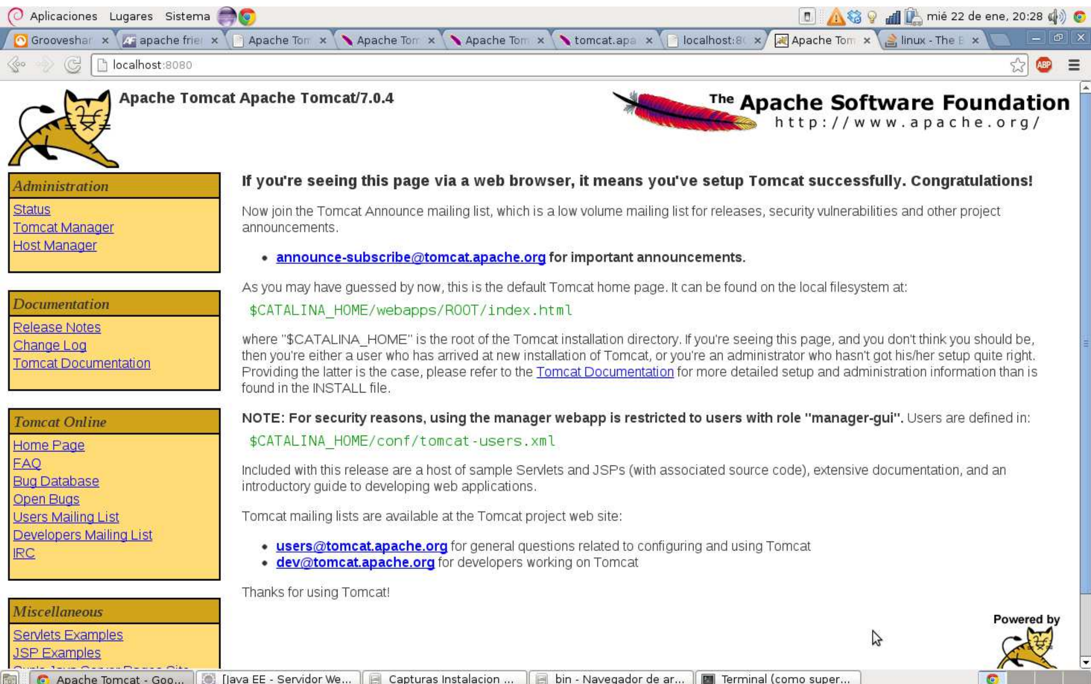


Ilustración 39 – Página de inicio de Tomcat

Tras esto tendremos que compilar la aplicación web (webapp) y exportarla en formato war, para poder ejecutarla dentro del servidor Tomcat. Esto lo podemos realizar fácilmente usando la opción de exportar en Eclipse [AÑADIR LA FOTO].

Una vez tengamos el war de la aplicación tendremos que incluirlo en el servidor. Para ello copiaremos el archivo en la carpeta *webapps*, dentro de la instalación de Tomcat.

Después tenemos que activar la aplicación, algo que se puede realizar mediante el Manager que viene incluido con Tomcat. Para poder ejecutarlo tendremos que añadir los permisos requeridos de ejecución del manager, que vienen desactivados por defecto por cuestiones de seguridad. Para ello tendremos que modificar el archivo tomcat-users.xml, que se incluye en la carpeta conf y dejar los siguientes roles (podemos cambiar el usuario y contraseña por los que queramos):

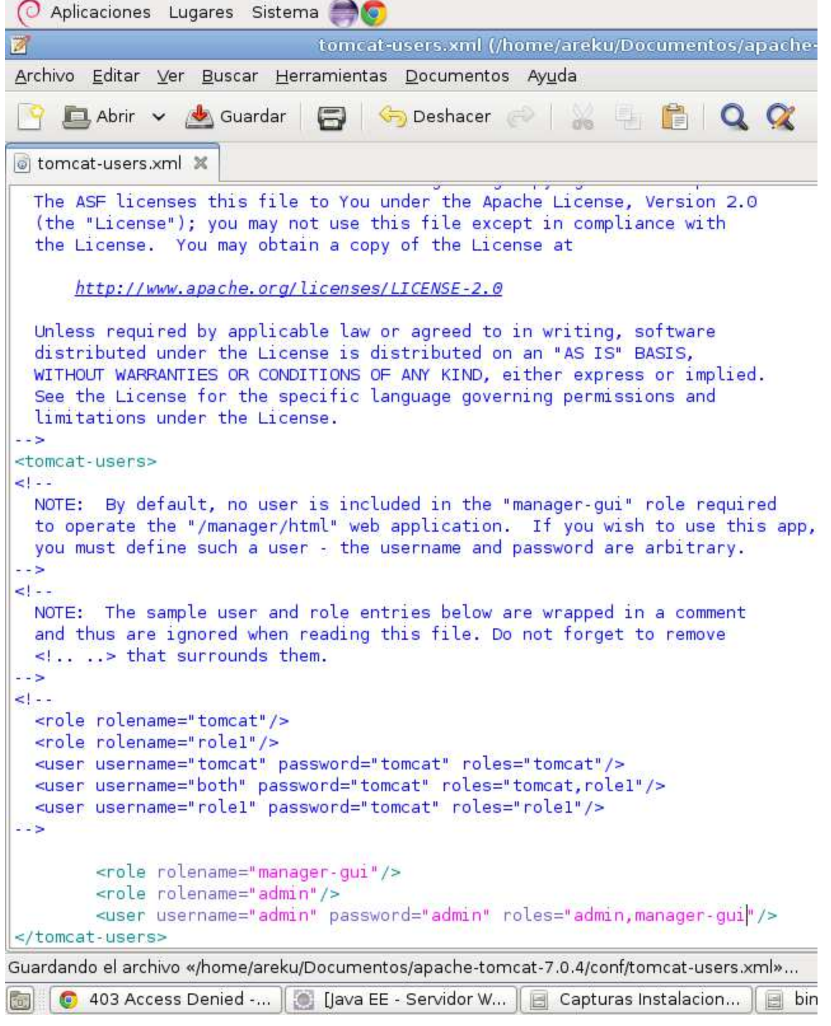


Ilustración 40 – Configuraciones de usuarios en Tomcat

Después de esto podemos acceder al Manager (localhost:8080/manager) con el usuario y contraseña especificados. Aquí podemos observar todas las webapps instaladas y podremos gestionarlas, siendo lo más importante arrancarlas y pararlas. Dentro de la lista observaremos la webapp que incluimos antes en el servidor (pfc.war) y podremos arrancarla o pararla. Por defecto, cuando arrancamos el servidor Tomcat, todas sus aplicaciones web instaladas se ejecutarán, por lo tanto no será necesario hacer esto cada vez que iniciemos el servidor.

Ilustración 41 – Gestor de aplicaciones Web de Tomcat

Para acceder a la aplicación web del proyecto (pfc.war) si está arrancado es:

localhost:8080/pfc

7.1.4. Uso del servidor

Al acceder al servidor web desde un navegador encontraremos un mapa de Google Maps junto con una barra de control.

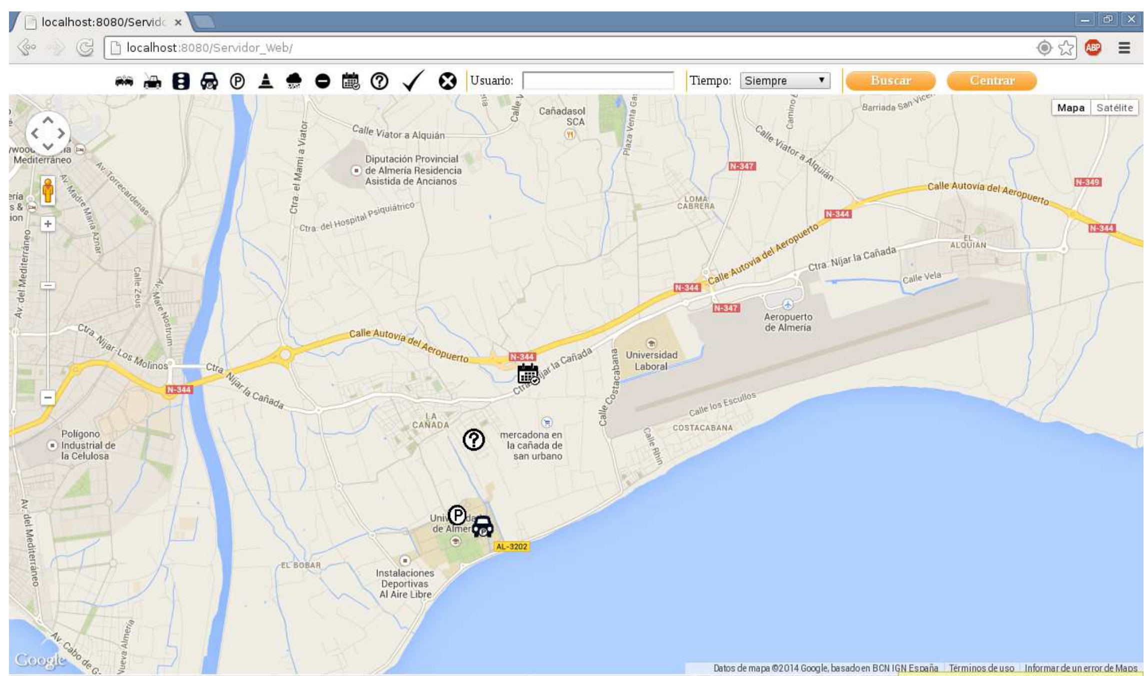


Ilustración 42 – Servidor web del sistema de incidencias

Por defecto se comenzará mostrando todas las incidencias obtenidas hoy (en un rango desde ahora hasta hace 24 horas). Además, recibiremos en tiempo real aquellas incidencias nuevas recibidas mientras estamos conectados, y se actualizará el mapa en concordancia.

Al pasar el ratón por encima de una incidencia, podremos ver sus detalles, tales como el usuario que envía la incidencia, el día, la hora y el texto de observaciones.

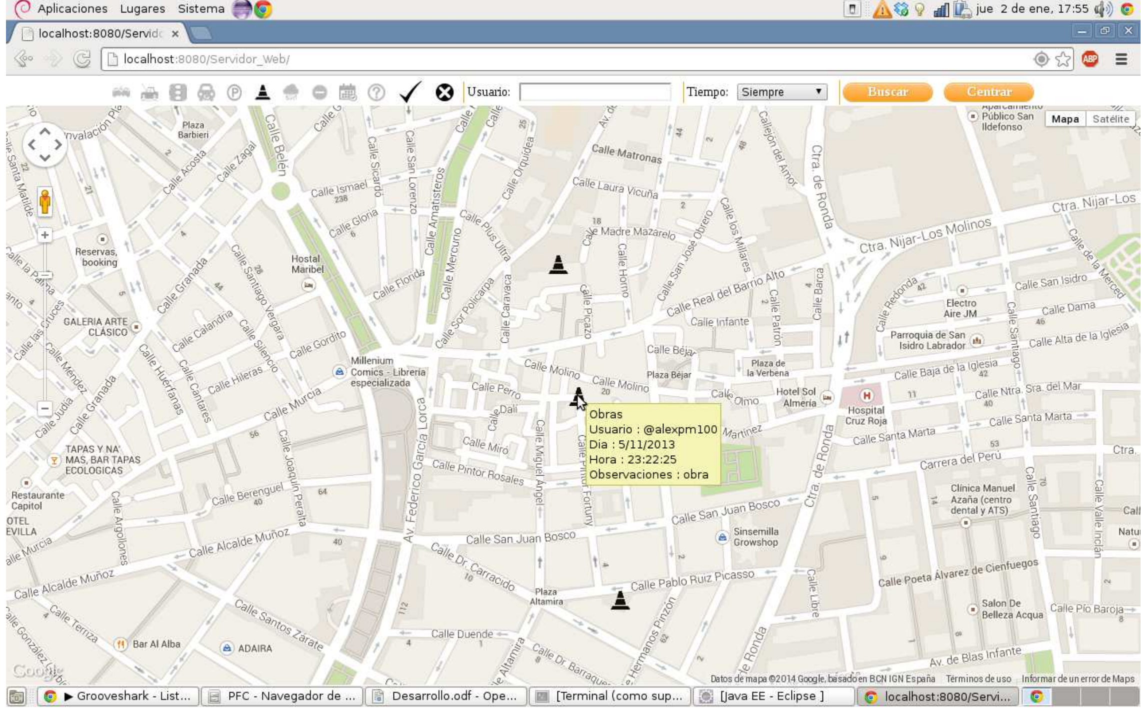


Ilustración 43 – Detalles de una incidencia

El usuario podrá seleccionar de la barra de herramientas cómo filtrar los resultados de las incidencias, pudiendo seleccionar las siguientes opciones:

- **Tipo de incidencia**: A la izquierda se encuentra un selector de tipos de incidencia. Se podrán marcar o desmarcar aquellos tipos de incidencia que se quieran mostrar. Además, existen botones para marcar todas o desmarcar todas.
- **Usuario**: Se podrán filtrar las incidencias por nombre de usuario de Twitter. En caso de dejar el campo en blanco no se tendrá en cuenta y se mostrarán incidencias de todos los usuarios.
- **Tiempo**: Se filtrarán las incidencias por rangos temporales. Se podrán elegir los siguientes:
    - **Hoy**: Desde ahora hasta hace 24 horas.
    - **Semana**: Desde ahora hasta hace 7 días.
    - **Mes**: Desde ahora hasta hace 30 días.
    - **Siempre**: Se recogen incidencias de cualquier fecha.

Una vez realizados los cambios en los campos de filtrado el usuario podrá pulsar el botón “Buscar”. En este caso se realizará la carga de las nuevas incidencias, que sustituirán a las anteriores en el mapa. Además, se renovará el sistema de recogida de nuevas incidencias para que sólo se actualicen en el mapa aquellas que cumplan las características.

Además, el usuario podrá centrar el mapa haciendo click en el botón “Centrar”, lo cual hará que se ajuste la posición del mapa y el zoom para acomodar a todos los marcadores de incidencia en el mapa. También podrá manipular el mapa como de costumbre, moviéndolo o cambiando el zoom, de forma manual.

7.2. Diagramas de la aplicación

7.2.1. Casos de uso

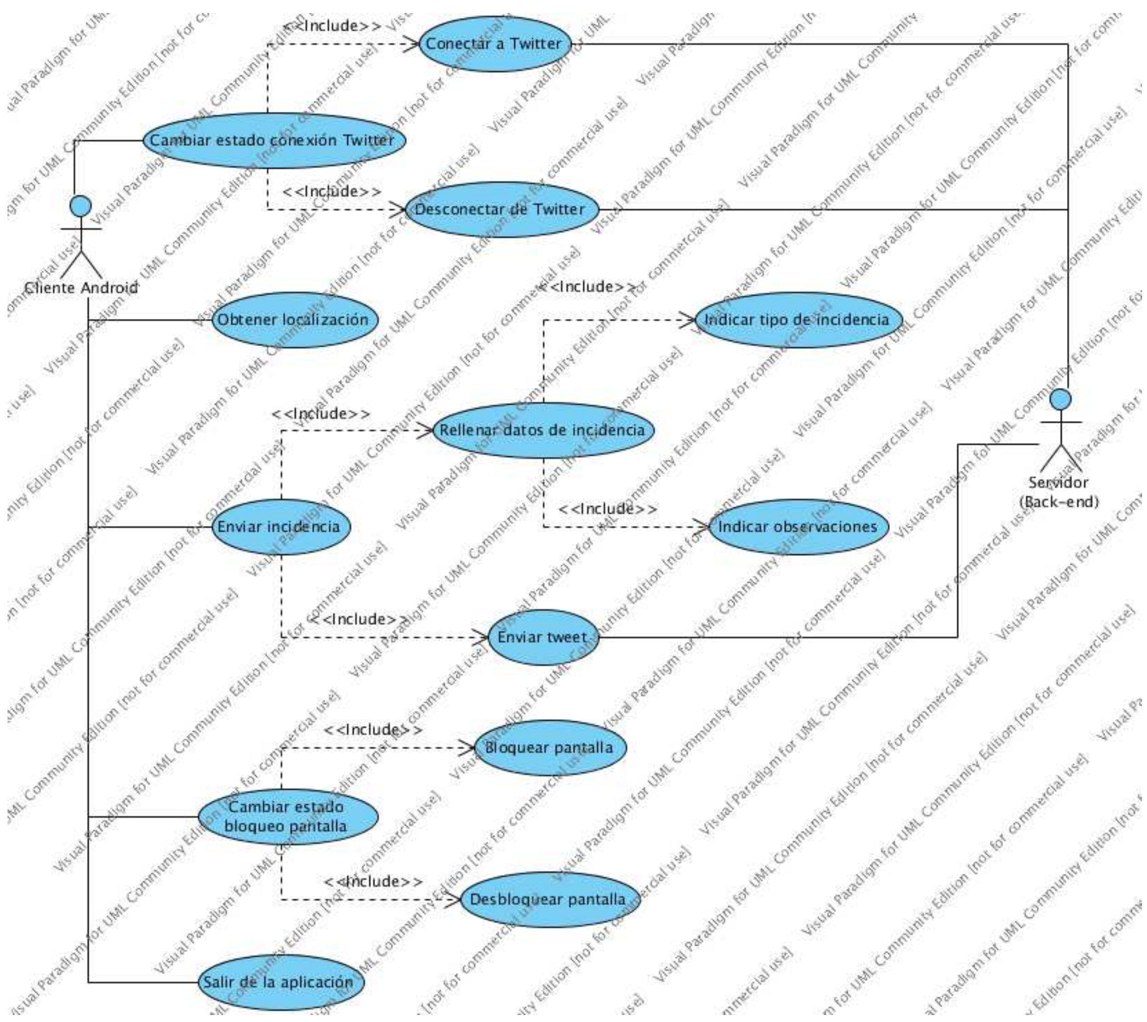


Ilustración 44 – Diagrama de casos de uso de la aplicación.

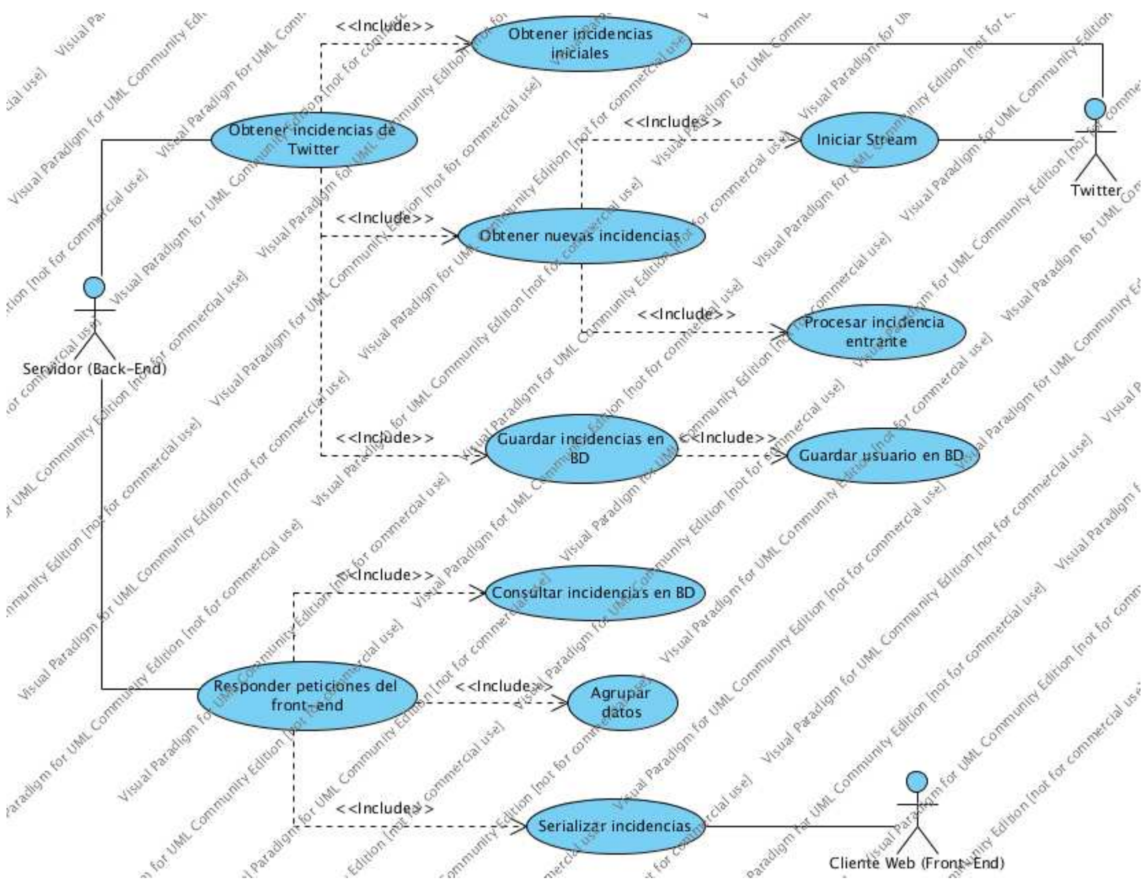


Ilustración 45 – Diagrama de casos de uso del back-end

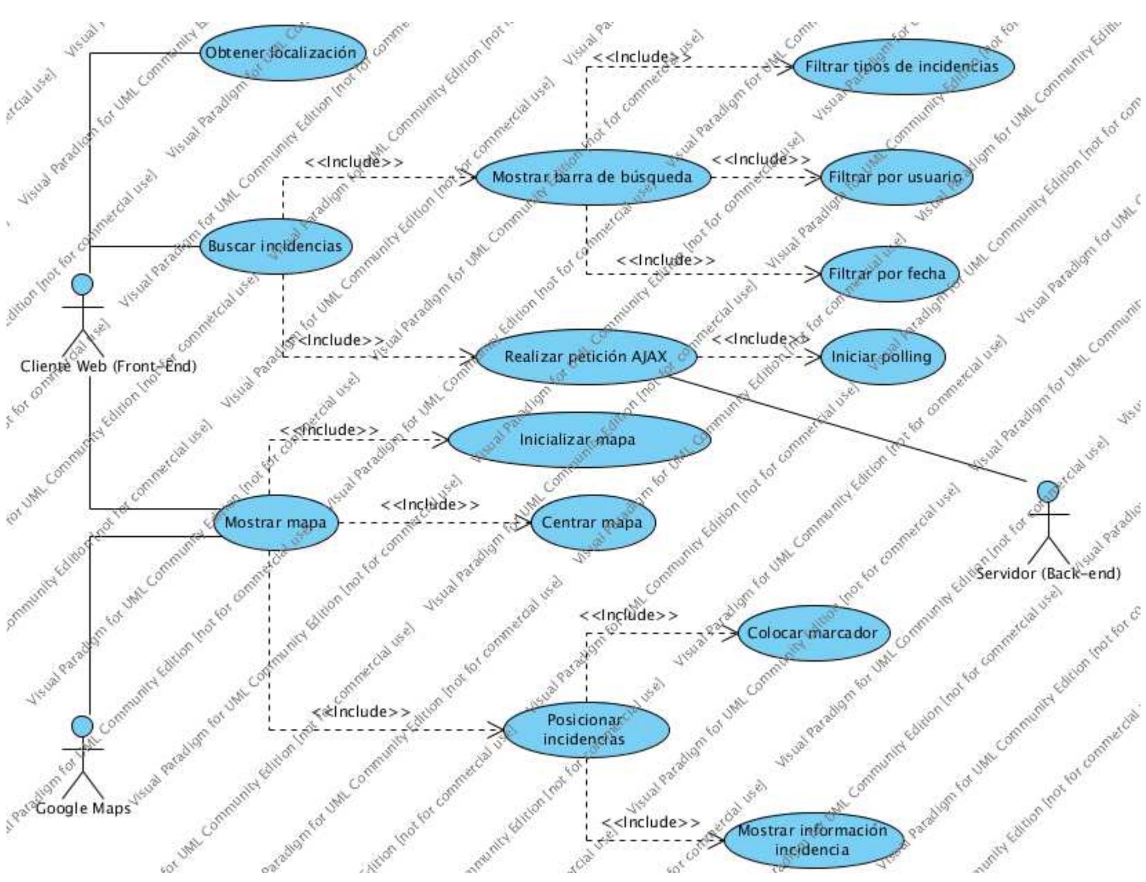


Ilustración 46 – Diagrama de casos de uso del front-end

7.2.2. Clases

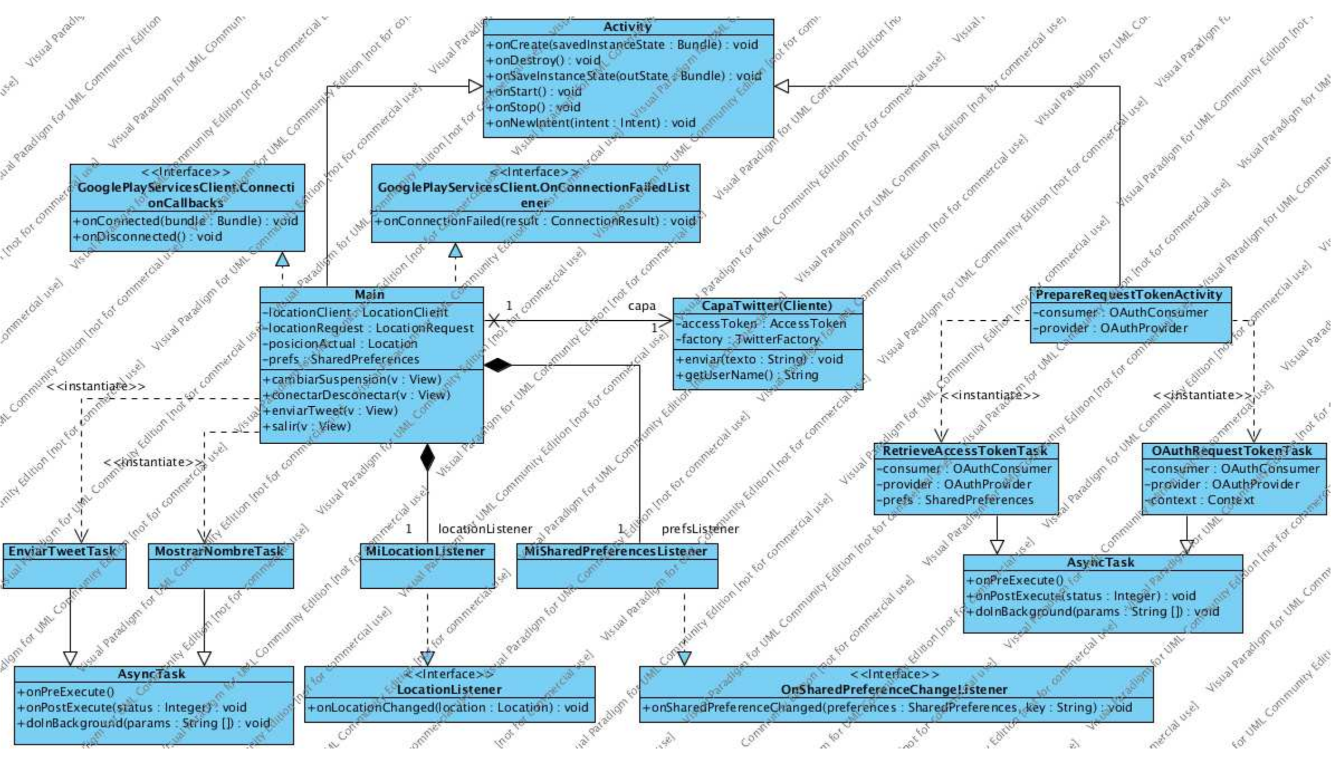


Ilustración 47 – Diagrama de clases del cliente Android

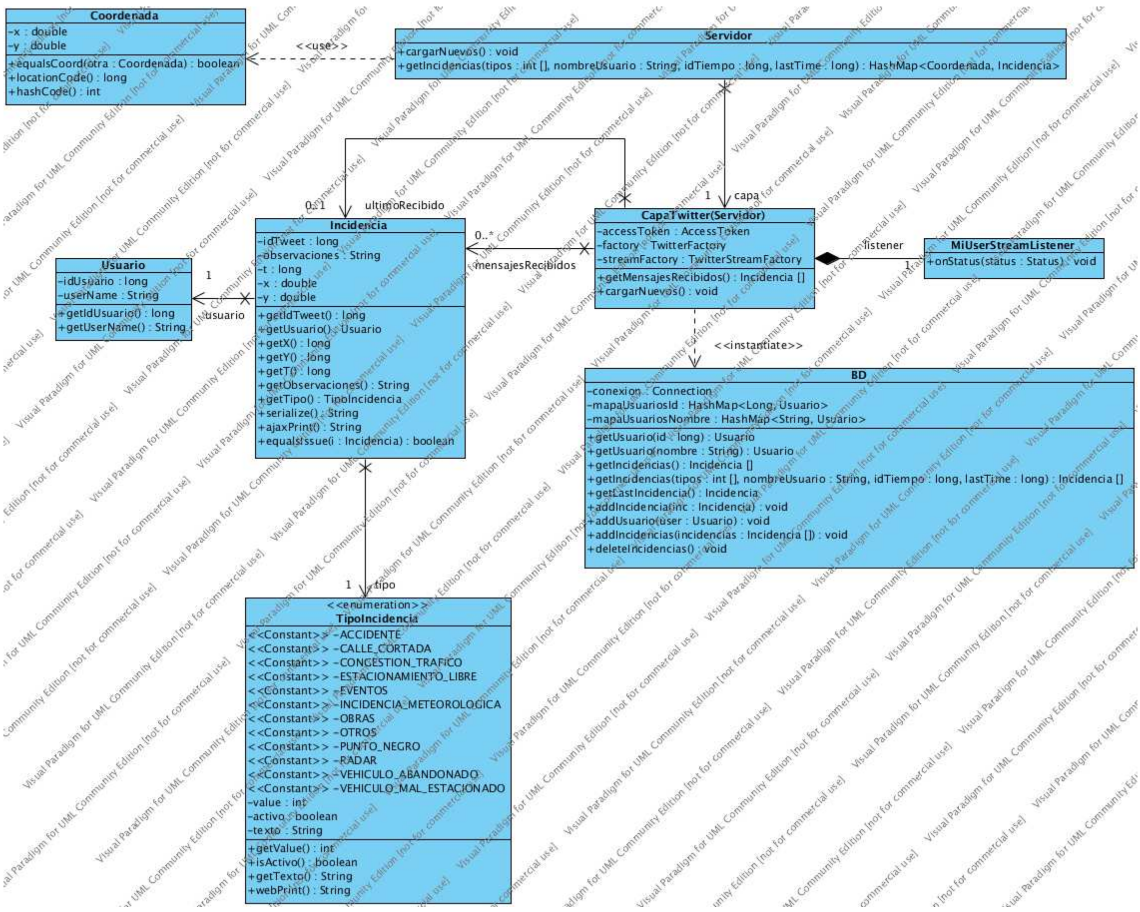


Ilustración 48 – Diagrama de clases del back-end

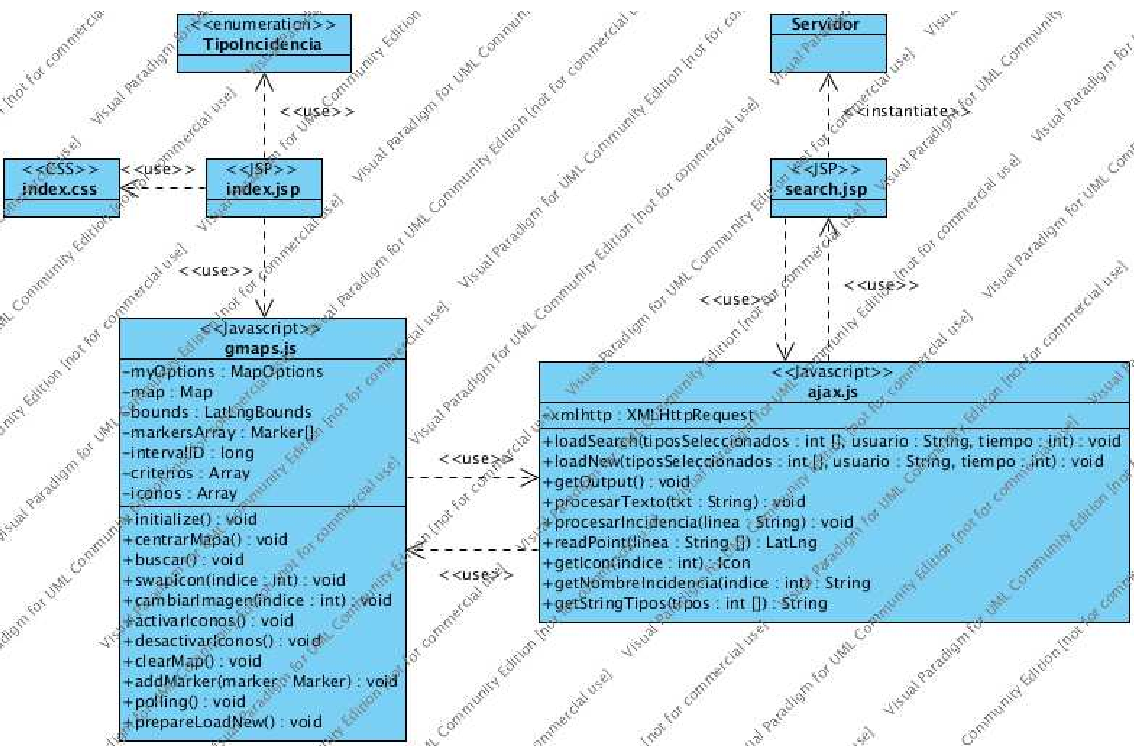


Ilustración 49 – Diagrama de clases del front-end

7.2.3. Secuencia

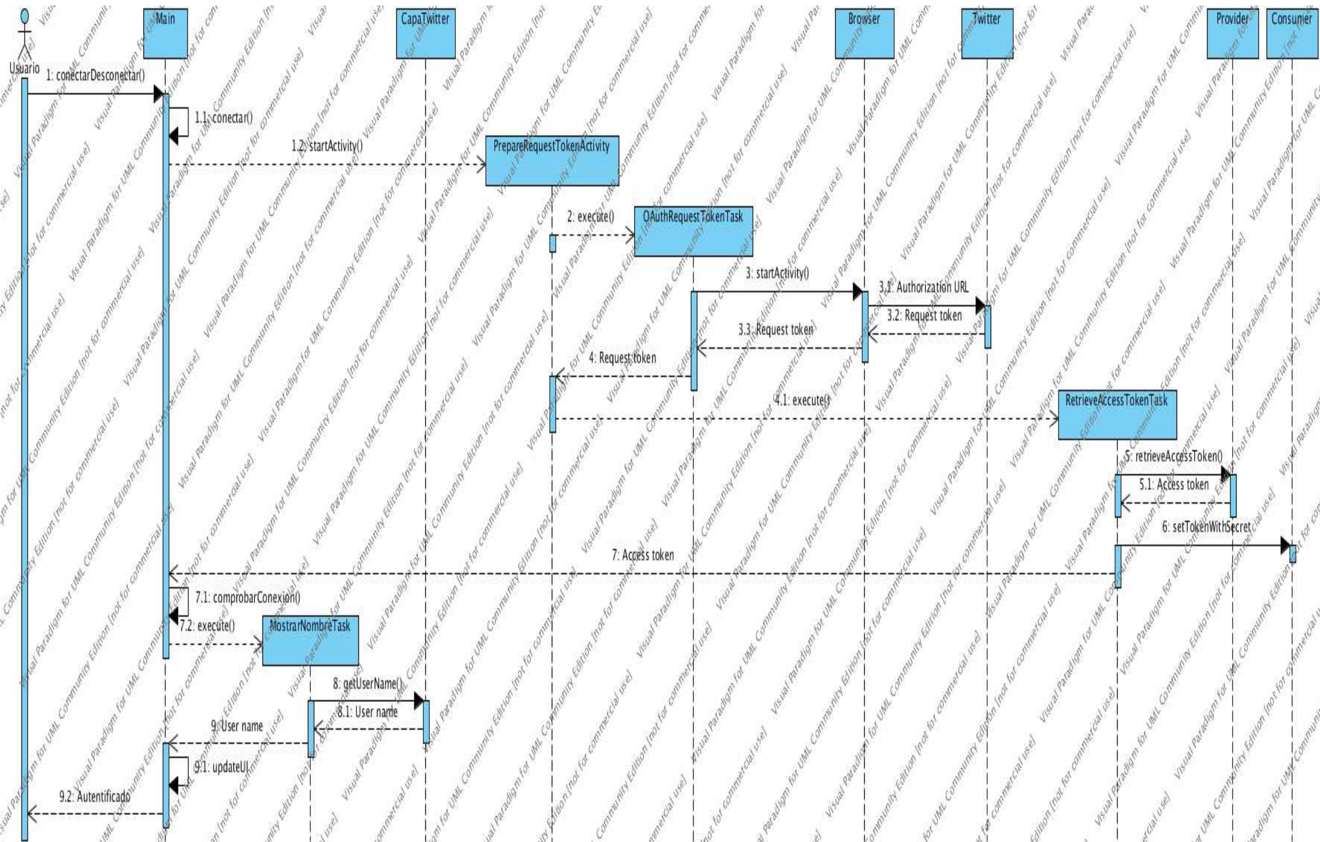


Ilustración 50 – Diagrama de secuencia de la autentificación en Twitter del cliente

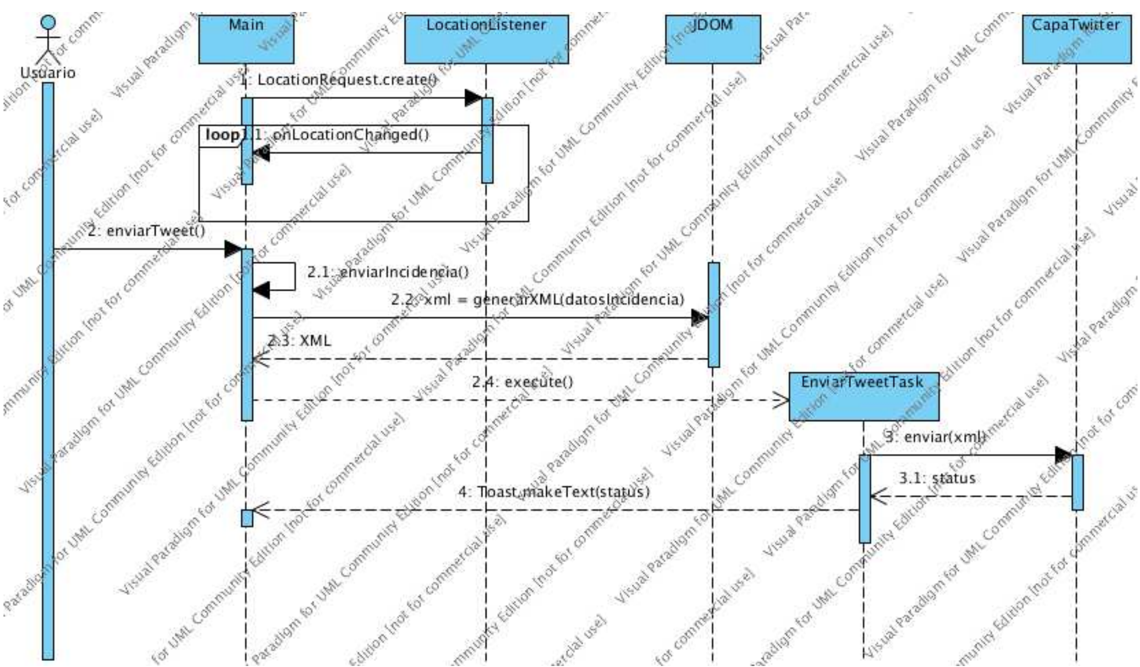


Ilustración 51 – Diagrama de secuencia del envío de incidencias del cliente

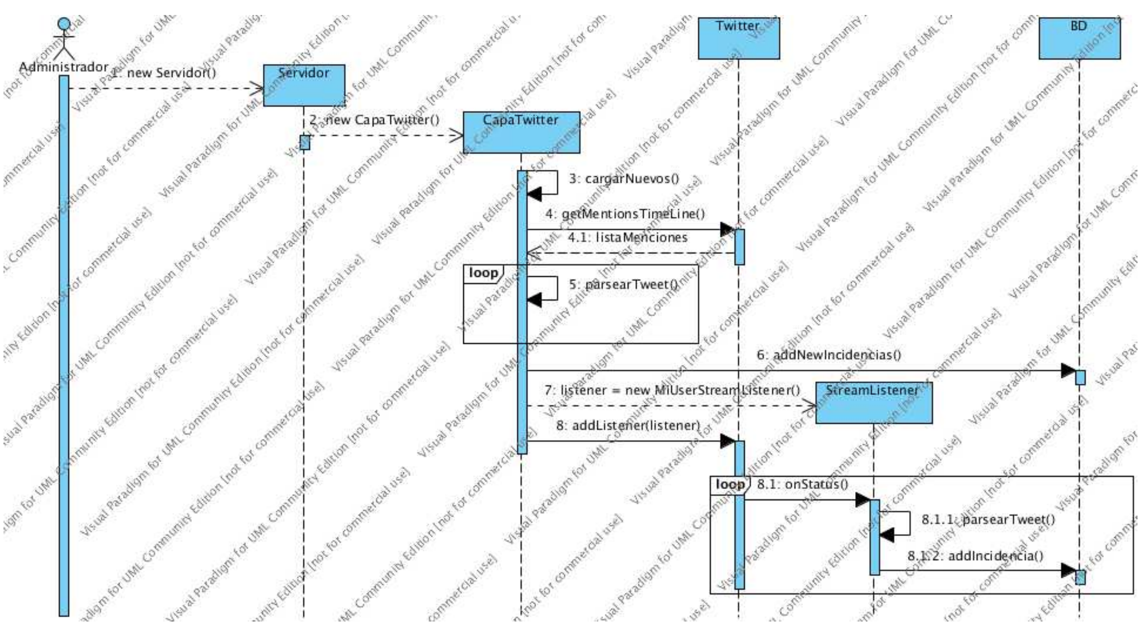


Ilustración 52 – Diagrama de secuencia de la obtención de nuevas incidencias del servidor

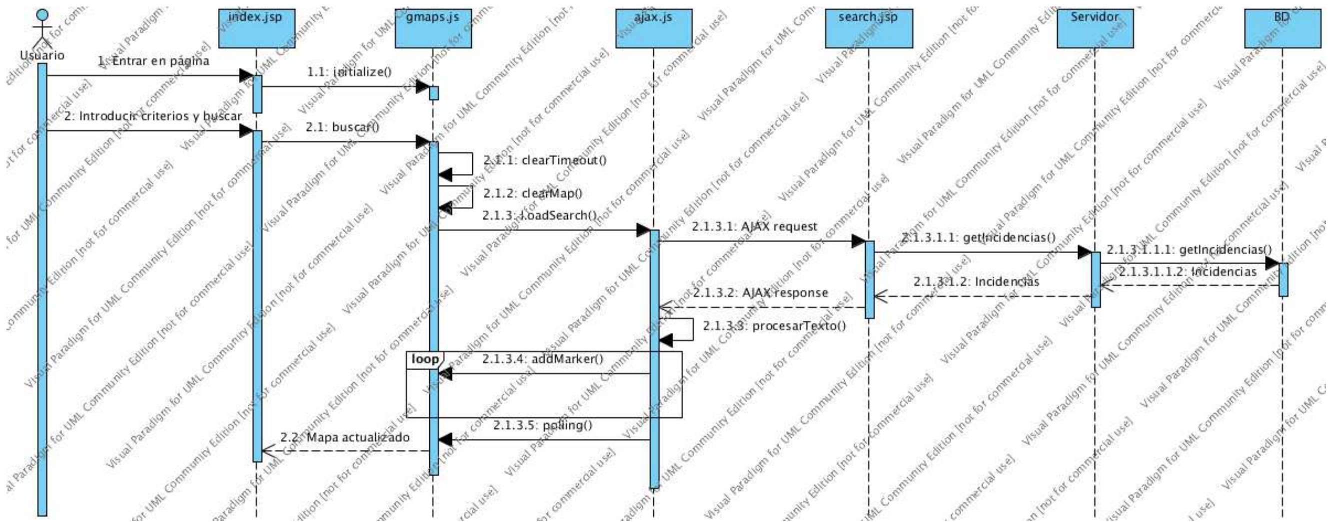


Ilustración 53 – Diagrama de secuencia de la búsqueda de incidencias en el sistema

7.2.4. Estados

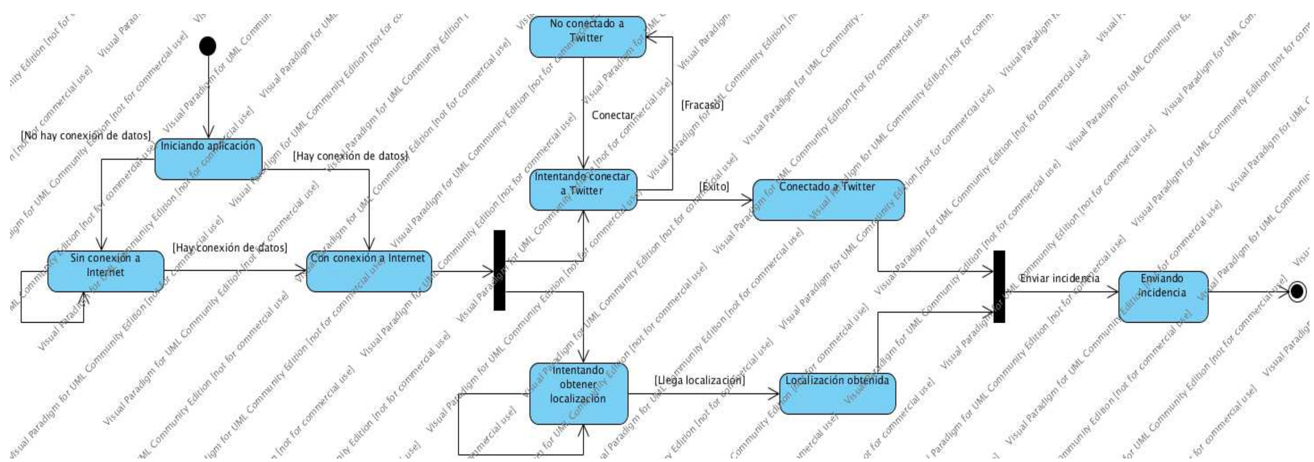


Ilustración 54 – Diagrama de estados del cliente Android

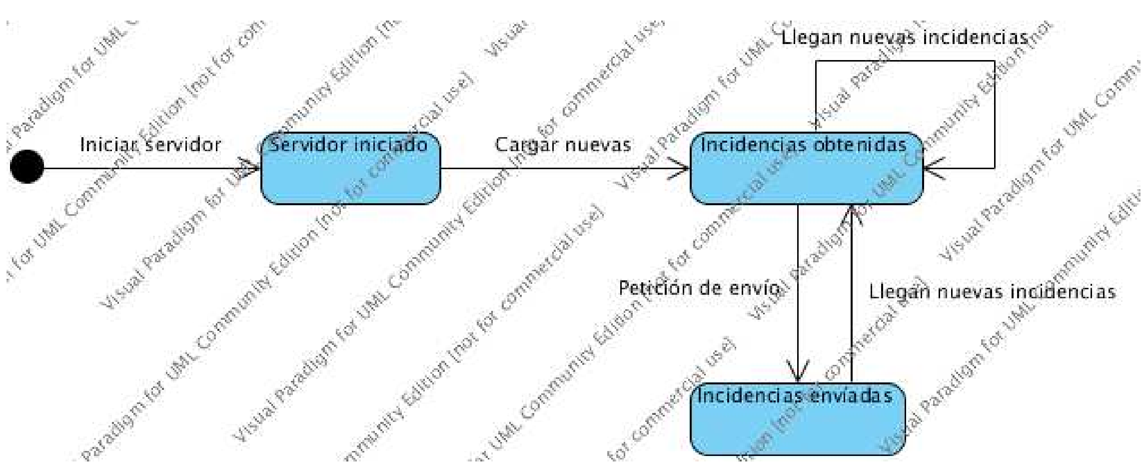


Ilustración 55 – Diagrama de estados del servidor (back-end)

7.3. Código fuente

7.3.1. Cliente

- **Main.java**

```
package com.pfc.client;


import java.util.Date;


import oauth.signpost.OAuth;


import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.output.Format;
import org.jdom2.output.XMLOutputter;
```

```
import twitter4j.TwitterException;
import twitter4j.auth.AccessToken;
import android.app.Activity;
import android.app.PendingIntent;
import android.app.PendingIntent.CanceledException;
import android.app.ProgressDialog;
import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.content.SharedPreferences.Editor;
import
android.content.SharedPreferences.OnSharedPreferenceChangeListener;
import android.location.Location;
import android.media.AudioManager;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.os.AsyncTask;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.util.SparseIntArray;
import android.view.Menu;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.Spinner;
import android.widget.TextView;
import android.widget.Toast;

import com.google.android.gms.common.ConnectionResult;
```

```
import
com.google.android.gms.common.GooglePlayServicesClient.ConnectionCallb
acks;
import
com.google.android.gms.common.GooglePlayServicesClient.OnConnectionFa
iledListener;
import com.google.android.gms.common.GooglePlayServicesUtil;
import com.google.android.gms.location.LocationClient;
import com.google.android.gms.location.LocationListener;
import com.google.android.gms.location.LocationRequest;
import com.pfc.core.CapaTwitter;
import com.pfc.oauth.Constants;


/**
 * Actividad principal de la aplicación Android. Gestiona el comportamiento
 * básico de la interfaz y el comportamiento con otros módulos.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class Main extends Activity implements
        ConnectionCallbacks,OnConnectionFailedListener {


        /**
         * Código de petición de conexión a Google Play Services.
         */
        private static final int GOOGLE_PLAY_REQUEST = 1;


        /**
         * Clave para guardar en preferencias si estamos conectados
         * a Twitter.
         */
        private static final String CONNECTED = "connected";
```

```
/**
 * Clave para guardar en preferencias el nombre de usuario en
 * Twitter.
 */
private static final String TWITTER_NAME = "twitter_name";

/**
 * Clave para guardar en preferencias si el check de bloquear
 * está activo
 */
private static final String CHECK_BLOCK = "check_block";

/**
 * Número de milisegundos de un segundo.
 */
private static final long SEGUNDO = 1000;
/**
 * Número de milisegundos de un minuto.
 */
private static final long MINUTO = SEGUNDO * 60;

/**
 * Número de milisegundos de una hora.
 */
private static final long HORA = MINUTO * 60;

/**
 * Número de milisegundos de un día.
 */
private static final long DIA = HORA * 24;
```

```
/**
 * Mapa de valores para el spinner.
 * Determina para cada indice del array que valor se debe enviar.
 */
private static SparseIntArray mapaValores;

/**
 * Instancia de la librería de twitter.
 */
private CapaTwitter capa;

/**
 * Preferencias de la aplicación (guarda datos para la activity).
 */
private SharedPreferences prefs;

/**
 * La posición actual del cliente.
 */
private Location posicionActual;

/**
 * Para saber si nos encontramos conectados a Twitter o no.
 */
private boolean conectado;

/**
 * Para saber si podemos conectarnos a Google Play Services
 */
private boolean servicesOK;

/**
```

```
 * Para saber si debemos bloquear la suspensión.
 */
private boolean bloquear;

/**
 * Cliente para la obtención de localizaciones actualizadas.
 */
private LocationClient locationClient;

/**
 * Petición de actualización para las localizaciones.
 * Se definirá como de alta precisión.
 */
private LocationRequest locationRequest;

/**
 * Listener para recibir las peticiones de actualización de
 * localización
 */
private LocationListener locationListener;

/**
 * Listener para escuchar cambios en las variables de
 * SharedPreferences (tokens de OAuth).
 */
private OnSharedPreferenceChangeListener prefsListener;

/**
 * Diálogo de progreso para mostrar el envío de tweet.
 */
private ProgressDialog twitterDialog;
```

```
    /**
     * Selector de incidencias.
     */
    private Spinner spinnerIncidencias;

    static {
            mapaValores = new SparseIntArray(10);
            mapaValores.put(0,0);
            mapaValores.put(1,1);
            mapaValores.put(2,2);
            mapaValores.put(3,3);
            mapaValores.put(4,4);
            mapaValores.put(5,7);
            mapaValores.put(6,8);
            mapaValores.put(7,9);
            mapaValores.put(8,10);
            mapaValores.put(9,11);
    }

    @Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  setVolumeControlStream(AudioManager.STREAM_MUSIC);

  twitterDialog = new ProgressDialog(this);
  twitterDialog.setIndeterminate(true);
            twitterDialog.setCancelable(false);
            twitterDialog.setTitle(R.string.enviandoTweet);
            twitterDialog.setMessage(getString(R.string.pleaseWait));
```

```
		prefs = PreferenceManager.getDefaultSharedPreferences(this);
	prefsListener = new MiSharedPreferencesListener();
	prefs.registerOnSharedPreferenceChangeListener(prefsListener);

		posicionActual = null;

		spinnerIncidencias = (Spinner) findViewById(
				R.id.spinnerIncidencias);
		ArrayAdapter<CharSequence> adapter = ArrayAdapter.

	createFromResource(this,R.array.arrayIncidencias,

	android.R.layout.simple_spinner_item);
		adapter.setDropDownViewResource(

	android.R.layout.simple_spinner_dropdown_item);
		spinnerIncidencias.setAdapter(adapter);

		if (savedInstanceState==null) {
			bloquear = false;
		}
		else {
			bloquear = savedInstanceState.
					getBoolean(CHECK_BLOCK,false);
		}
		CheckBox checkBloquear =
(CheckBox)findViewById(R.id.checkSuspension);
	checkBloquear.setChecked(bloquear);

	servicesOK = false;
	int gPlay = GooglePlayServicesUtil.isGooglePlayServicesAvailable(
		this);
	if (gPlay!=ConnectionResult.SUCCESS) {
```

```
  PendingIntent pi = GooglePlayServicesUtil.
                    getErrorPendingIntent(
                    gPlay,this,
                    GOOGLE_PLAY_REQUEST);
  try {
                              pi.send();
                    } catch (CanceledException e) {
                              e.printStackTrace();
                    }
}
else {
  servicesOK = true;
}

locationClient = new LocationClient(this, this, this);
locationRequest = LocationRequest.create();
locationRequest.setPriority(
          LocationRequest.PRIORITY_HIGH_ACCURACY);
locationRequest.setInterval(5000);
locationListener = new MiLocationListener();

conectado = prefs.getBoolean(CONNECTED,false);

if (!hayInternet())
  conectado = false;

if(!conectado)
  comprobarConexion();
else {
  AccessToken token = cargarAccessToken();
  if (token == null) {
          desconectar();
```

```
        }
        else {
                TextView labelNombre=(TextView)findViewById(
                                R.id.labelNombre);
                String nombre = prefs.getString(TWITTER_NAME,
                                getString(R.string.conectando));
                labelNombre.setText(nombre);
                capa = new CapaTwitter(token);
        }
    }

}

@Override
public void onDestroy() {
        if (locationClient.isConnected()) {
                locationClient.removeLocationUpdates(locationListener);
        }
        locationClient.disconnect();
        prefs.unregisterOnSharedPreferenceChangeListener(prefsListener);

        super.onDestroy();
}

@Override
protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);

        outState.putBoolean(CHECK_BLOCK, bloquear);
}

@Override
```

```
public void onStop() {

    if (locationClient.isConnected()) {
            locationClient.removeLocationUpdates(locationListener);
    }
    locationClient.disconnect();
    super.onStop();
}

@Override
public void onStart() {
    super.onStart();

    locationClient.connect();
    if (!conectado)
            comprobarConexion();
}

@Override
    protected void onActivityResult(int requestCode,
                    int resultCode, Intent data) {
            switch (requestCode) {
            case GOOGLE_PLAY_REQUEST:
                    if (resultCode == Activity.RESULT_OK) {
                            int gPlay = GooglePlayServicesUtil.

    isGooglePlayServicesAvailable(this);
                            if (gPlay == ConnectionResult.SUCCESS)
                                    servicesOK = true;
                            else {
                                    servicesOK = false;
                            }
```

```
                    }
                    else {
                            servicesOK = false;
                    }
                    break;
            }
    }

@Override
public boolean onCreateOptionsMenu(Menu menu) {
  return true;
}

@Override
    public void onConnectionFailed(ConnectionResult result) {
    posicionActual = null;
    updateUI();
    }

    @Override
    public void onConnected(Bundle bundle) {
            locationClient.requestLocationUpdates(locationRequest,
                    locationListener);
            posicionActual = locationClient.getLastLocation();

            updateUI();
    }

    @Override
    public void onDisconnected() {
            updateUI();
    }
```

```
/**
 * Enviamos un tweet con la incidencia al pulsar el botón.
 * @param v El botón que activa el evento.
 */
public void enviarTweet(View v) {
    if (posicionActual==null) {
        Toast.makeText(this,R.string.locationUnknown,
                Toast.LENGTH_LONG).show();
    }
    int indice = spinnerIncidencias.getSelectedItemPosition();
    TextView areaTexto = (TextView)findViewById(R.id.areaTexto);
    String texto = areaTexto.getText().toString();
    enviarIncidencia(indice,texto);
}

    /**
     * Funcion que sirve para conectar o desconectar, según el estado
actual.
     * @param v La vista (botón) que lanza la acción.
     */
    public void conectarDesconectar(View v) {
        if (conectado)
            desconectar();
        else
            conectar();
    }

    /**
     * Salimos de la aplicación.
     * @param v La vista (botón) que lanza la acción.
     */
```

```
public void salir(View v) {
        finish();
}

/**
 * Cambiamos el modo de suspensión.
 * @param v La vista (botón) que lanza la acción.
 */
public void cambiarSuspension(View v) {
        updateSuspension(!bloquear);
}

/**
 * Enviamos una incidencia a traves de Twitter.
 * @param tipo El tipo de incidencia (indice de array antes de mapear)
 * @param texto El texto a enviar (puede ser vacío).
 */
private void enviarIncidencia(int tipo,String texto) {
        Element element=new Element("inc");
element.setAttribute("x",String.valueOf(
                posicionActual.getLatitude()));
element.setAttribute("y",String.valueOf(
                posicionActual.getLongitude()));
element.setAttribute("t",String.valueOf(new Date().getTime()));

int codigo = mapaValores.valueAt(tipo);
element.setAttribute("i",String.valueOf(codigo));
if (texto.length()>0) {
        element.setAttribute("text",texto);
}
Document document=new Document(element);
Format format=Format.getCompactFormat();
```

```
	format.setOmitEncoding(true);
	format.setOmitDeclaration(true);
	XMLOutputter outputter=new XMLOutputter(format);
	String xml=outputter.outputString(document);
	String tweet = Constants.ACCOUNT_NAME + " " + xml;

	new EnviarTweetTask().execute(tweet);
	}

	/**
	 * Actualizamos el estado de suspensión del teléfono.
	 * @param block El estado final de bloqueo (bloqueado / no
bloqueado)
	 */
	private void updateSuspension(boolean block) {
		bloquear = block;

		View principal = findViewById(R.id.layoutPrincipal);
		principal.setKeepScreenOn(block);

		CheckBox checkSuspension = (CheckBox)findViewById(
				R.id.checkSuspension);
		checkSuspension.setChecked(block);
	}

	/**
	 * Cargamos el AccessToken de las preferencias (memoria persistente).
	 * @return El AccessToken del usuario. Si no tiene devuelve null.
	 */
	private AccessToken cargarAccessToken() {
		String token = prefs.getString(OAuth.OAUTH_TOKEN, "");
		String secret = prefs.getString(OAuth.OAUTH_TOKEN_SECRET,
"");
```

```
        if (token.equals(""))
            return null;
        if (secret.equals(""))
            return null;
        return new AccessToken(token,secret);
    }

    /**
     * Desconectamos el usuario de Twitter.
     * Para ello se borra el AccessToken del usuario.
     * @param v La vista (botón) que lanza la acción.
     */
    private void desconectar() {
        TextView labelNombre=(TextView)findViewById(
                R.id.labelNombre);
        labelNombre.setText(R.string.noConectado);
        conectado = false;

        Editor editor = prefs.edit();
        editor.remove(OAuth.OAUTH_TOKEN);
        editor.remove(OAuth.OAUTH_TOKEN_SECRET);
        editor.putBoolean(Constants.OAUTH_SUCCESS,false);
        editor.putBoolean(CONNECTED,conectado);
        editor.commit();

        updateUI();

    }

    /**
     * Conectamos al usuario a su cuenta de Twitter.
     * Para ello se lanza la actividad que gestiona la conexión a Twitter
```

```
     * y obtenemos un nuevo AccesToken.
     * @param v La vista (botón) que lanza la acción.
     */
    private void conectar() {
        Intent i = new Intent(this,PrepareRequestTokenActivity.class);
        startActivity(i);

        comprobarConexion();
    }

    /**
     * Actualizamos la interfaz gráfica
     */
    private void updateUI() {
        Button botonTweet=(Button)findViewById(R.id.botonTweet);
        Button
botonConectar=(Button)findViewById(R.id.botonConectar);
        TextView
labelLocation=(TextView)findViewById(R.id.labelLocation);

        String locationStr;
        if (!servicesOK) {
            locationStr = getString(R.string.servicesDesactivados);
        }
        else if (!locationClient.isConnected()) {
            if (locationClient.isConnecting()) {
                locationStr = getString(R.string.conectando);
            }
            else {
                locationStr = getString(R.string.desconectado);
            }
        }
        else if (posicionActual == null) {
```

```
			locationStr = getString(R.string.recibiendoDatos);
		}
		else {
			long locationTime = posicionActual.getTime();
			long time = System.currentTimeMillis();
			long delta = time - locationTime;
			int locationId;
			long param;
			if (delta < SEGUNDO) {
				param = -1;
				locationId = R.string.datosRecibidosYa;
			}
			else if (delta < MINUTO) {
				param = delta / 1000;
				locationId = R.string.datosRecibidosSegundos;
			}
			else if (delta < HORA) {
				param = delta / (1000*60);
				locationId = R.string.datosRecibidosMinutos;
			}
			else if (delta < DIA) {
				param = delta / (1000*60*60);
				locationId = R.string.datosRecibidosHoras;
			}
			else {
				param = delta / (1000*60*60*24);
				locationId = R.string.datosRecibidosDias;
			}
			locationStr = getString(locationId,param);


		}
```

```
                labelLocation.setText(locationStr);


                if (!hayInternet())
                        conectado = false;


                if (conectado) {
                        botonConectar.setText(R.string.botonDesconectar);
                }
                else {
                        botonConectar.setText(R.string.botonConectar);
                }


                botonTweet.setEnabled(conectado &&
locationClient.isConnected()
                                && posicionActual!=null);
        }


   /**
    * Comprobamos si la conexión con Twitter es correcta.
    * Se actualizarán los componentes que sean necesarios.
    */
   private void comprobarConexion() {
        TextView labelNombre=(TextView)findViewById(
                R.id.labelNombre);
        if (!hayInternet()) {
                labelNombre.setText(R.string.noInternet);
        }
        else {
                AccessToken token = cargarAccessToken();
         if (token != null || prefs.getBoolean(
                        Constants.OAUTH_SUCCESS,false)) {
           labelNombre.setText(R.string.conectando);
```

```
                capa=new CapaTwitter(token);
                new MostrarNombreTask().execute((Void[])null);
        }
        else {
            labelNombre.setText(R.string.noConectado);
            conectado=false;
            Editor e = prefs.edit();
            e.putBoolean(CONNECTED, conectado);
            e.commit();
        }
      }
   updateUI();
}

/**
 * Comprobamos si tenemos o no conexión a Internet.
 * @return true si hay conexión, false si no.
 */
     private boolean hayInternet() {
                ConnectivityManager cm = (ConnectivityManager)

     getSystemService(Context.CONNECTIVITY_SERVICE);
                NetworkInfo netInfo = cm.getActiveNetworkInfo();
                if (netInfo != null && netInfo.isConnected()) {
                        return true;
                }
                return false;
     }

/**
 * Tarea asíncrona encargada de enviar un tweet en segundo plano.
 * @author Alejandro Pérez Manzano
```

```
 *
 */
private class EnviarTweetTask extends
AsyncTask<String, Void, Integer> {

    /**
     * Si el tweet se envía correctamente.
     */
    private static final int SUCCESS = 0;
    /**
     * Si se han excedido el límite de peticiones a Twitter.
     */
    private static final int LIMIT_EXCEEDED = 1;
    /**
     * Si ha ocurrido algún otro error al enviar el Tweet.
     */
    private static final int OTHER_ERROR = 100;

    /**
     * Puntero al boton de enviar tweet.
     */
    private Button boton;

    /**
     * Creamos la nueva AsyncTask para enviar un tweet.
     */
    public EnviarTweetTask() {
                boton = (Button)findViewById(R.id.botonTweet);
        }

        @Override
        protected Integer doInBackground(String... params) {
```

```
        try {
            capa.enviar(params[0]);
            return SUCCESS;
        } catch (TwitterException e) {
            if (e.exceededRateLimitation()) {
                return LIMIT_EXCEEDED;
            }
            else {
                return OTHER_ERROR;
            }
        }
    }

    @Override
    protected void onPreExecute() {
        boton.setEnabled(false);
        twitterDialog.show();
    }

    @Override
    protected void onPostExecute(Integer status) {
        twitterDialog.dismiss();
        boton.setEnabled(true);
        int idMensaje = R.string.tweetError;
        switch(status) {
        case SUCCESS:
            idMensaje = R.string.tweetExito;
            break;
        case LIMIT_EXCEEDED:
            idMensaje = R.string.tweetLimiteExcedido;
            break;
        case OTHER_ERROR:
```

```
                        idMensaje = R.string.tweetError;
                        break;
                    }
                    Toast toast = Toast.makeText(getApplicationContext(),
                            idMensaje,Toast.LENGTH_LONG);
                toast.show();
                }

}

/**
 * Tarea asíncrona encargada de comprobar si las credenciales
 * de Twitter son correctas y obtener el nombre del usuario.
 *
 * @author Alejandro Pérez Manzano
 *
 */
private class MostrarNombreTask
extends AsyncTask<Void, Void, String> {

            @Override
            protected String doInBackground(Void... params) {
                    String nombre = capa.getUserName();
                    return nombre;
            }

            @Override
            protected void onPostExecute(String nombre) {
                    if (nombre.equals(CapaTwitter.LIMITE_EXCEDIDO)) {
                            nombre=getString(R.string.noConectado);
                            conectado = false;
```

```
    Toast.makeText(getBaseContext(),R.string.tweetLimiteExcedido,
                                        Toast.LENGTH_LONG).show();
            }
            else if (nombre.equals("")) {
                nombre=getString(R.string.noConectado);
                conectado = false;
            }
            else {
                nombre="@"+nombre;
                conectado = true;
            }

            Editor e = prefs.edit();
            e.putBoolean(CONNECTED,conectado);
            e.putString(TWITTER_NAME,nombre);
            e.commit();

            updateUI();
            TextView labelNombre=(TextView)findViewById(
                        R.id.labelNombre);
            labelNombre.setText(nombre);
        }

}

/**
 * Implementación propia del LocationListener, para procesar
 * las actualizaciones de localización recibidas.
 *
 * @author Alejandro Pérez Manzano
 *
```

```
 */
private class MiLocationListener implements LocationListener {

        @Override
        public void onLocationChanged(Location location) {
                posicionActual = location;
                updateUI();
        }

}

/**
 * Clase privada para controlar los cambios de preferencias.
 * Se utiliza para gestionar la vuelta desde la obtención
 * de permisos OAuth de Twitter.
 *
 * @author Alejandro Pérez Manzano
 *
 */
    private class MiSharedPreferencesListener implements
                OnSharedPreferenceChangeListener {

        @Override
        public void onSharedPreferenceChanged(
                        SharedPreferences sharedPreferences, String
key) {
                if (key.equals(Constants.OAUTH_SUCCESS)
                                && sharedPreferences.getBoolean(key,
false)) {
                        comprobarConexion();
                }

        }
```

```
        }

}
```

- **PrepareRequestTokenActivity.java**

```
package com.pfc.client;


import com.pfc.oauth.Constants;
import com.pfc.oauth.OAuthRequestTokenTask;


import oauth.signpost.OAuth;
import oauth.signpost.OAuthConsumer;
import oauth.signpost.OAuthProvider;
import oauth.signpost.commonshttp.CommonsHttpOAuthConsumer;
import oauth.signpost.commonshttp.CommonsHttpOAuthProvider;
import android.app.Activity;
import android.content.Intent;
import android.content.SharedPreferences;
import android.content.SharedPreferences.Editor;
import android.net.Uri;
import android.os.AsyncTask;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.util.Log;


/**
 * Prepara un OAuthConsumer y un OAuthProvider
 *
 * El OAuthConsumer se configura con el par de claves pública/privada
 * El OAuthProvider se configura a partir de los endpoints de la conexión.
 *
 * Tras ello se ejecuta la tarea asíncrona (AsyncTask) encargada de realizar
```

```
 * la petición al servidor de Twitter.
 *
 * Una vez recibida la respuesta por la AsyncTask, se devuelve el foco
 * a esta actividad para que procese dicha respuesta.
 *
 */
public class PrepareRequestTokenActivity extends Activity {

	/**
	 * Etiqueta para la información de log.
	 */
	private static final String TAG = "PFC";

	/**
	 * Consumidor de OAuth.
	 */
    private OAuthConsumer consumer;

    /**
     * Proveedor de OAuth.
     */
    private OAuthProvider provider;

	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
	try {
		this.consumer = new CommonsHttpOAuthConsumer(
					Constants.CONSUMER_KEY,
Constants.CONSUMER_SECRET);
	    this.provider = new CommonsHttpOAuthProvider(
				Constants.REQUEST_URL,Constants.ACCESS_URL,
```

```
                            Constants.AUTHORIZE_URL);
        } catch (Exception e) {
                Log.e(TAG, "Error creating consumer / provider",e);
                }

      Log.i(TAG, "Starting task to retrieve request token.");
                new
OAuthRequestTokenTask(this,consumer,provider).execute();
        }

        /**
         * Función callback que es llamada desde la tarea asíncrona para
         * procesar la respuesta y obtener de ella el AccessToken.
         */
        @Override
        public void onNewIntent(Intent intent) {
                super.onNewIntent(intent);
                SharedPreferences prefs = PreferenceManager.
                                getDefaultSharedPreferences(this);
                final Uri uri = intent.getData();
                if (uri != null && uri.getScheme().equals(
                                Constants.OAUTH_CALLBACK_SCHEME)) {
                        Log.i(TAG, "Callback received : " + uri);
                        Log.i(TAG, "Retrieving Access Token");
                        new
RetrieveAccessTokenTask(consumer,provider,prefs).
                                execute(uri);
                        finish();
                }
        }

        /**
         * Tarea asíncrona para reconstruir el AccessToken a partir de la
```

```
 * respuesta ofrecida por Twitter tras la petición de autentificación
 * OAuth.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class RetrieveAccessTokenTask extends
        AsyncTask<Uri,Void,Void> {

        /**
         * Proveedor de OAuth.
         */
        private OAuthProvider provider;
        /**
         * Consumidor de OAuth.
         */
        private OAuthConsumer consumer;
        /**
         * Preferencias de la aplicación (para guardar datos)
         */
        private SharedPreferences prefs;

        /**
         * Constructor de la tarea.
         * @param consumer Consumidor
         * @param provider Proveedor
         * @param prefs Preferencias
         */
        public RetrieveAccessTokenTask(OAuthConsumer consumer,
                        OAuthProvider provider, SharedPreferences
prefs) {
                this.consumer = consumer;
```

```
			this.provider = provider;
			this.prefs=prefs;
		}



		/**
		 * Obtiene el token de acceso a partir de la respuesta de Twitter.
		 */
		@Override
		protected Void doInBackground(Uri...params) {
			final Uri uri = params[0];
			final String oauth_verifier = uri.getQueryParameter(
					OAuth.OAUTH_VERIFIER);

			try {
				provider.retrieveAccessToken(consumer, oauth_verifier);

				String token = consumer.getToken();
				String secret = consumer.getTokenSecret();

				consumer.setTokenWithSecret(token, secret);

				Log.i(TAG, "OAuth - Access Token Retrieved");

			} catch (Exception e) {
				Log.e(TAG, "OAuth - Access Token Retrieval Error", e);
			}

			return null;
		}
```

```
		/**
		 * Guardamos en las preferencias de la aplicación el token de
		 * acceso.
		 */
		@Override
		protected void onPostExecute(Void result) {
			final Editor edit = prefs.edit();
			edit.putString(OAuth.OAUTH_TOKEN,
consumer.getToken());
			edit.putString(OAuth.OAUTH_TOKEN_SECRET,
					consumer.getTokenSecret());
			edit.putBoolean(Constants.OAUTH_SUCCESS, true);
			Log.d("PFC","Token escrito a preferences");
			edit.commit();
		}

	}

}
```

- **CapaTwitter.java**

```
package com.pfc.core;


import com.pfc.oauth.Constants;


import android.util.Log;
import twitter4j.Twitter;
import twitter4j.TwitterException;
import twitter4j.TwitterFactory;
import twitter4j.auth.AccessToken;
import twitter4j.conf.ConfigurationBuilder;
```

```
/**
 * Capa de comunicación entre Cliente y Servidor a traves de
 * mensajes Twitter.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class CapaTwitter {


	/**
	 * La clase Factory con la configuración requerida.
	 */
	private TwitterFactory factory;

	/**
	 * El token de acceso del usuario con OAuth.
	 */
	private AccessToken accessToken;

	/**
	 * Constante para definir que hemos excedido el número de consultas.
	 */
	public static final String LIMITE_EXCEDIDO = "LIMITE_EXCEDIDO";

	/**
	 * Creamos la capa de conexión a Twitter a partir de un
	 * token de acceso obtenido previamente con OAuth.
	 *
	 * @param accessToken El token de acceso.
	 */
	public CapaTwitter(AccessToken accessToken) {
		ConfigurationBuilder cb=new ConfigurationBuilder();
```

```
		cb.setOAuthConsumerKey(Constants.CONSUMER_KEY);
		cb.setOAuthConsumerSecret(Constants.CONSUMER_SECRET);
		cb.setDebugEnabled(false);
		cb.setUseSSL(true);
		factory = new TwitterFactory(cb.build());
		this.accessToken=accessToken;
	}

	/**
	 * Enviamos el texto a la cuenta de la aplicación.
	 *
	 * @param texto El texto a enviar.
	 *
	 * @throws TwitterException Si ha ocurrido algun error.
	 */
	public void enviar(String texto) throws TwitterException {
		Twitter twitter = factory.getInstance(accessToken);
		twitter.updateStatus(texto);
	}

	/**
	 * Devolvemos el nombre del usuario conectado.
	 * También se utiliza para comprobar si estamos realmente
	 * conectados o no.
	 *
	 * @return El nombre de usuario, o "" si no estamos conectados.
	 */
	public String getUserName() {
		Twitter twitter=factory.getInstance(accessToken);
		try {
			return twitter.getScreenName();
		} catch (IllegalStateException e) {
```

```
                        Log.e("PFC","Error al conectar a nuestra cuenta",e);
                        return "";
                } catch (TwitterException e) {
                        if (e.exceededRateLimitation()) {
                                Log.e("PFC","Limite excedido",e);
                                return LIMITE_EXCEDIDO;
                        }
                        else {
                                Log.e("PFC","Error al conectar a nuestra
cuenta",e);
                                return "";
                        }
                }
        }

}
```

- **Constants.java**

```
package com.pfc.oauth;

/**
 * Constantes necesarias para la autentificación OAuth.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class Constants {

        /**
         * Clave pública de mi aplicación.
         */
        public static final String CONSUMER_KEY =
                        "PR08GIXxfaVsVbg0wYjA";
```

```
/**
 * Clave privada de mi aplicación.
 */
public static final String CONSUMER_SECRET=
                "sEt8ceVWIpFONWU3p8iAgWVgpadIwHlqRCD3ow08";

/**
 * URL del endpoint de petición OAuth.
 */
public static final String REQUEST_URL =
                "https://api.twitter.com/oauth/request_token";
/**
 * URL del endpoint de acceso OAuth.
 */
public static final String ACCESS_URL =
                "https://api.twitter.com/oauth/access_token";
/**
 * URL del endpoint de autorización OAuth.
 */
public static final String AUTHORIZE_URL =
                "https://api.twitter.com/oauth/authorize";

/**
 * Etiqueta que define la aplicación de callback
 */
public static final String OAUTH_CALLBACK_SCHEME =
                "x-oauth-twitter";
/**
 * Host que recibirá el callback.
 */
public static final String OAUTH_CALLBACK_HOST =
                "callback";
```

```
	/**
	 * URL completa de callback.
	 */
	public static final String OAUTH_CALLBACK_URL =
			OAUTH_CALLBACK_SCHEME + "://" +
OAUTH_CALLBACK_HOST;

	/**
	 * Nombre de la cuenta de la aplicación.
	 */
	public static final String ACCOUNT_NAME =
			"@alexpm100";

	/**
	 * Clave para comprobar si ha habido éxito en la operación
	 * de autorización OAuth,
	 */
	public static final String OAUTH_SUCCESS =
			"oauth_success";
}
```

- **OauthRequestTokenTask.java**

```
package com.pfc.oauth;


import oauth.signpost.OAuthConsumer;
import oauth.signpost.OAuthProvider;
import android.content.Context;
import android.content.Intent;
import android.net.Uri;
import android.os.AsyncTask;
import android.util.Log;


/**
```

```
 * Una tarea asíncrona para gestionar la comunicación
 * aplicación-usuario-Twitter para la obtención de un token de acceso.
 *
 * Para ello se requerirá de conexión mediante un navegador Web a
 * la página de Twitter para que el usuario introduzca sus credenciales
 * (usuario/contraseña).
 *
 */
public class OAuthRequestTokenTask extends AsyncTask<Void, Void, Void> {

	/**
	 * Etiqueta para la información de Log.
	 */
	final String TAG = getClass().getName();
	/**
	 * Contexto de la aplicación Android. Se usa para poder lanzar el
	 * navegador Web.
	 */
	private Context context;
	/**
	 * Proveedor de OAuth.
	 */
	private OAuthProvider provider;
	/**
	 * Consumidor de OAuth.
	 */
	private OAuthConsumer consumer;

	/**
	 *
	 * Creamos la tarea asíncrona.
	 *
```

```
	 * @param context El contexto de la aplicación.
	 * @param consumer El consumidor OAuth.
	 * @param provider El proveedor OAuth.
	 */
	public OAuthRequestTokenTask(Context context,
			OAuthConsumer consumer,OAuthProvider provider) {
		this.context = context;
		this.consumer = consumer;
		this.provider = provider;
	}

	/**
	 * Obtiene el request token para poder realizar la petición.
	 * Posteriormente, se usa ese request token para abrir la página
	 * de petición de Twitter al usuario y así obtener su AccessToken.
	 */
	@Override
	protected Void doInBackground(Void... params) {

		try {
			Log.i(TAG, "Retrieving request token from Google
servers");
			final String url =
provider.retrieveRequestToken(consumer,
					Constants.OAUTH_CALLBACK_URL);
			Log.i(TAG,"Popping a browser with the authorize URL :
" + url);
			Intent intent = new
Intent(Intent.ACTION_VIEW,Uri.parse(url)).

	setFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP |

	Intent.FLAG_ACTIVITY_NO_HISTORY |
```

```
        Intent.FLAG_FROM_BACKGROUND);
                        context.startActivity(intent);
                } catch (Exception e) {
                        Log.e(TAG, "Error during OAUth retrieve request
token", e);
                }

                return null;
        }

}
```

- **layout/main.xml**

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

    <LinearLayout
        android:id="@+id/layoutPrincipal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="15dp"
        android:focusable="true"
          android:focusableInTouchMode="true">

        <TableLayout
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:paddingBottom="20dp" >
```

```
<TableRow>

   <TextView
      android:id="@+id/labelNombreLeft"
      android:layout_width="wrap_content"
      android:layout_height="fill_parent"
      android:paddingRight="20sp"
      android:text="@string/twitter"
      android:textSize="15sp"
      android:textStyle="bold" />

   <TextView
      android:id="@+id/labelNombre"
      android:layout_width="wrap_content"
      android:layout_height="fill_parent"
      android:textSize="15sp" />
</TableRow>

<TableRow>

   <TextView
      android:id="@+id/labelLocationLeft"
      android:layout_width="wrap_content"
      android:layout_height="fill_parent"
      android:paddingRight="20sp"
      android:text="@string/location"
      android:textSize="15sp"
      android:textStyle="bold" />

   <TextView
      android:id="@+id/labelLocation"
```

```
        android:layout_width="wrap_content"

        android:layout_height="fill_parent"

        android:textSize="15sp" />

    </TableRow>

</TableLayout>


<TextView

    android:layout_width="fill_parent"

    android:layout_height="wrap_content"

    android:layout_marginBottom="5dp"

    android:gravity="left"

    android:text="@string/labelIncidencia"

    android:textSize="15sp"

    android:textStyle="bold" />


<Spinner

    android:id="@+id/spinnerIncidencias"

    android:layout_width="fill_parent"

    android:layout_height="wrap_content"

    android:spinnerMode="dropdown"/>


<TextView

    android:layout_width="fill_parent"

    android:layout_height="wrap_content"

    android:layout_marginBottom="5dp"

    android:gravity="left"

    android:text="@string/labelTexto"

    android:textSize="15sp"

    android:textStyle="bold" />


<EditText

    android:id="@+id/areaTexto"
```

```
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="20dp"
    android:gravity="top|left"
    android:inputType="textMultiLine"
    android:lines="2"
    android:maxLength="50"
    android:scrollHorizontally="false" />

  <TableLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" >

    <TableRow>

      <Button
        android:id="@+id/botonTweet"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:enabled="false"
        android:onClick="enviarTweet"
        android:text="@string/botonTweet" />
    </TableRow>

    <TableRow>

      <Button
        android:id="@+id/botonConectar"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="2"
```

```
            android:onClick="conectarDesconectar"
            android:text="@string/botonConectar" />

        <Button
            android:id="@+id/botonSalir"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="2"
            android:onClick="salir"
            android:text="@string/botonSalir" />
    </TableRow>
  </TableLayout>

  <LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:orientation="horizontal" >

    <CheckBox
        android:id="@+id/checkSuspension"
        android:layout_width="0dip"
        android:layout_height="wrap_content"
        android:layout_marginRight="5dp"
        android:layout_weight="2"
        android:checked="false"
        android:onClick="cambiarSuspension"
        android:text="@string/checkSuspension" />
  </LinearLayout>

</LinearLayout>
```

```
</ScrollView>
```

- **layout-land/main.xml**

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <LinearLayout
        android:id="@+id/layoutPrincipal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="15dp"
        android:focusable="true"
          android:focusableInTouchMode="true">

        <LinearLayout
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_marginBottom="20dp"
            android:baselineAligned="false"
            android:orientation="horizontal" >

            <TableLayout
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:layout_weight="2" >

                <TableRow>

                    <TextView
                        android:id="@+id/labelNombreLeft"
```

```
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:paddingRight="20sp"
            android:text="@string/twitter"
            android:textSize="15sp"
            android:textStyle="bold" />

        <TextView
            android:id="@+id/labelNombre"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:textSize="15sp" />
    </TableRow>

    <TableRow>

        <TextView
            android:id="@+id/labelLocationLeft"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:paddingRight="20sp"
            android:text="@string/location"
            android:textSize="15sp"
            android:textStyle="bold" />

        <TextView
            android:id="@+id/labelLocation"
            android:layout_width="wrap_content"
            android:layout_height="fill_parent"
            android:textSize="15sp" />
    </TableRow>
</TableLayout>
```

```
<TableLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="2" >

    <TableRow>

        <Button
            android:id="@+id/botonTweet"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:enabled="false"
            android:onClick="enviarTweet"
            android:text="@string/botonTweet" />
    </TableRow>

    <TableRow>

        <Button
            android:id="@+id/botonConectar"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="2"
            android:onClick="conectarDesconectar"
            android:text="@string/botonConectar" />

        <Button
            android:id="@+id/botonSalir"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
```

```
            android:layout_weight="2"
            android:onClick="salir"
            android:text="@string/botonSalir" />
        </TableRow>
    </TableLayout>
</LinearLayout>

<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="5dp"
    android:gravity="left"
    android:text="@string/labelIncidencia"
    android:textSize="15sp"
    android:textStyle="bold" />

<Spinner
    android:id="@+id/spinnerIncidencias"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:spinnerMode="dropdown"/>

<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="5dp"
    android:gravity="left"
    android:text="@string/labelTexto"
    android:textSize="15sp"
    android:textStyle="bold" />

<EditText
```

```
        android:id="@+id/areaTexto"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="top|left"
        android:inputType="textMultiLine"
        android:lines="2"
        android:scrollHorizontally="false" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:orientation="horizontal">

        <CheckBox
            android:id="@+id/checkSuspension"
            android:layout_width="0dip"
            android:layout_height="wrap_content"
            android:layout_weight="2"
            android:checked="false"
            android:onClick="cambiarSuspension"
            android:text="@string/checkSuspension"
            android:layout_marginRight="5dp" />

    </LinearLayout>
  </LinearLayout>

</ScrollView>
```

- **AndroidManifest.xml**

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.pfc.client"
    android:versionCode="1"
```

```
    android:versionName="1.0" >


    <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="18" />


    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme"
        android:allowBackup="true">


        <activity
            android:name=".Main"
            android:label="@string/title_activity_main" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />


                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>


        <activity android:name=".PrepareRequestTokenActivity"
            android:launchMode="singleTask">
                        <intent-filter>
                                <action
android:name="android.intent.action.VIEW" />
                                <category
```

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

android:name="android.intent.category.BROWSABLE" />
                                        <data android:scheme="x-oauth-twitter"
                                          android:host="callback" />
                              </intent-filter>
                    </activity>



   </application>


</manifest>
```

7.3.2. Servidor

- **CapaTwitter.java**

```
package com.pfc.core;


import java.util.ArrayList;
import java.util.List;


import com.pfc.issues.Incidencia;
import com.pfc.issues.dao.BD;
import com.pfc.issues.exceptions.IssueException;


import twitter4j.DirectMessage;
import twitter4j.Paging;
import twitter4j.ResponseList;
import twitter4j.StallWarning;
import twitter4j.Status;
import twitter4j.StatusDeletionNotice;
import twitter4j.Twitter;
```

```
import twitter4j.TwitterException;
import twitter4j.TwitterFactory;
import twitter4j.TwitterStream;
import twitter4j.TwitterStreamFactory;
import twitter4j.User;
import twitter4j.UserList;
import twitter4j.UserStreamListener;
import twitter4j.auth.AccessToken;
import twitter4j.conf.Configuration;
import twitter4j.conf.ConfigurationBuilder;

/**
 * Capa de comunicación entre Cliente y Servidor a través de
 * mensajes Twitter.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class CapaTwitter {

	/**
	 * Clave pública de la aplicación
	 */
	private static final String CONSUMER_KEY =
			"PR08GIXxfaVsVbg0wYjA";

	/**
	 * Clave privada de la aplicación
	 */
	private static final String CONSUMER_SECRET =
			"sEt8ceVWIpFONWU3p8iAgWVgpadIwHlqRCD3ow08";
```

```
	/**
	 * Parte pública del token de acceso (para conectarse en nombre
	 * de la propia aplicación).
	 */
	private static final String ACCESS_TOKEN =
			"518776140-
F5zyD3QJmqfBQtCJgsxivkNV7k8UyUZxtTN1F0";

	/**
	 * Parte privada del token de acceso (para conectarse en nombre
	 * de la propia aplicación).
	 */
	private static final String ACCESS_TOKEN_SECRET =

	"vGNhU25lgCFyPY1B69yRFwvIvGRCFgTpTcAc8vmMHLY";

	/**
	 * Nombre de la cuenta de la aplicación.
	 */
	public static final String ACCOUNT_NAME =
			"@alexpm100";

	/**
	 * La clase Factory con la configuración requerida.
	 * Se utiliza para generar instancias de la interfaz de
	 * conexión a Twitter.
	 */
	private TwitterFactory factory;

	/**
	 * La clase StreamFactory con la configuración requerida.
	 * Similar al Factory normal, pero para conexiones en stream.
	 */
```

```
private TwitterStreamFactory streamFactory;

/**
 * El token de acceso del sistema con OAuth.
 */
private AccessToken accessToken;

/**
 * Lista de incidencias recibidas desde el último envío
 * al front-end.
 */
private List<Incidencia> mensajesRecibidos;

/**
 * Información de la última incidencia leída.
 */
private Incidencia ultimoRecibido;

/**
 * Listener para escuchar peticiones de Twitter mediante Stream,
 * en tiempo real.
 */
private MiUserStreamListener listener;



/**
 * Creamos una nueva conexión hacia Twitter, utilizando
 * las credenciales de conexión estáticas para el servidor
 * (es decir, las propias de la cuenta de la aplicación).
 *
 * Se realiza una carga inicial de las incidencias posteriores
 * a la última en la base de datos.
```

```
	 */
	public CapaTwitter() {
		ConfigurationBuilder cb=new ConfigurationBuilder();
		cb.setOAuthConsumerKey(CONSUMER_KEY);
		cb.setOAuthConsumerSecret(CONSUMER_SECRET);
		cb.setDebugEnabled(false);
		cb.setUseSSL(true);
		Configuration conf=cb.build();
		factory = new TwitterFactory(conf);
		streamFactory = new TwitterStreamFactory(conf);
		listener = new MiUserStreamListener();
		accessToken=new
AccessToken(ACCESS_TOKEN,ACCESS_TOKEN_SECRET);
		mensajesRecibidos=new ArrayList<Incidencia>();
		BD bd = BD.getInstance();
		ultimoRecibido = bd.getLastIncidencia();

		cargarNuevos();

		TwitterStream stream =
streamFactory.getInstance(accessToken);
		stream.addListener(listener);
		stream.user();
	}

	/**
	 * Devolvemos la lista de incidencias recibidas.
	 * Se limpia posteriormente la lista.
	 *
	 * @return La lista de mensajes recibidos no enviados al front-end.
	 */
	public List<Incidencia> getMensajesRecibidos() {
		List<Incidencia> clon =
```

```
                        new ArrayList<Incidencia>(mensajesRecibidos);
            mensajesRecibidos = new ArrayList<Incidencia>();
            return clon;
    }

    /**
     * Cargamos los mensajes a partir del ultimo guardado en BD.
     * Insertamos estos nuevos mensajes en la BD y los preparamos
     * para devolverlos al front-end.
     */
    public void cargarNuevos() {
            Twitter twitter = factory.getInstance(accessToken);
            Paging p = new Paging();
            p.setCount(200);
            if (ultimoRecibido != null) {
                    long id = ultimoRecibido.getIdTweet();
                    p.setSinceId(id);
            }

            List<Incidencia> listaMencionesFiltrada =
                        new ArrayList<Incidencia>();
            ResponseList<Status> listaMenciones;
            int pagina = 1;
            do {
                    p.setPage(pagina);
                    try {
                            listaMenciones =
twitter.getMentionsTimeline(p);
                    } catch (TwitterException e) {
                            e.printStackTrace();
                            return;
                    }
```

```
                        for (Status status : listaMenciones) {
                                Incidencia incidencia = parsearTweet(status);
                                if (incidencia != null) {
                                        listaMencionesFiltrada.add(incidencia);
                                        if (ultimoRecibido == null)
                                                ultimoRecibido = incidencia;
                                        else if (incidencia.getT() >
ultimoRecibido.getT())
                                                ultimoRecibido = incidencia;
                                }
                        }
                        pagina++;
                } while (listaMenciones.size() > 0);
                mensajesRecibidos.addAll(listaMencionesFiltrada);
                guardarMensajes();
        }

        /**
         * Guardamos los mensajes a BD.
         */
        private void guardarMensajes() {
                if (mensajesRecibidos.isEmpty()) {
                        return;
                }
                BD bd = BD.getInstance();
                bd.addNewIncidencias(mensajesRecibidos);
        }

        /**
         * Convertimos un tweet con el formato de la API de Twitter
         * a una incidencia de mi sistema.
         * @param status El tweet
```

```
 * @return La incidencia correspondiente, o null si no se
 * puede convertir.
 */
private Incidencia parsearTweet(Status status) {
	Incidencia incidencia = null;

	try {
		User user = status.getUser();
		incidencia = Incidencia.getFromXml(status.getId(),
				user.getId(),
				user.getScreenName(),
				status.getText());
	} catch (IssueException e) {
		incidencia = null;
	}

	return incidencia;
}

/**
 * Listener para procesar las peticiones llegadas en tiempo real.
 *
 * @author Alejandro Pérez Manzano
 *
 */
private class MiUserStreamListener implements UserStreamListener {

	@Override
	public void onException(Exception ex) {
		ex.printStackTrace();
	}
```

```
@Override
public void onTrackLimitationNotice(int arg0) {
}

@Override
public void onStatus(Status status) {
	Incidencia incidencia = parsearTweet(status);
	if (incidencia != null) {
		mensajesRecibidos.add(incidencia);
		if (incidencia.getT() > ultimoRecibido.getT())
			ultimoRecibido = incidencia;
		BD bd = BD.getInstance();
		bd.addIncidencia(incidencia);
	}
}

@Override
public void onScrubGeo(long arg0, long arg1) {
}

@Override
public void onDeletionNotice(StatusDeletionNotice arg0) {
}

@Override
public void onUserProfileUpdate(User updatedUser) {
}

@Override
public void onUserListUpdate(User listOwner, UserList list) {
}
```

```
@Override
public void onUserListUnsubscription(User subscriber,
        User listOwner,UserList list) {
}

@Override
public void onUserListSubscription(User subscriber,
        User listOwner,UserList list) {
}

@Override
public void onUserListMemberDeletion(User deletedMember,
        User listOwner,UserList list) {
}

@Override
public void onUserListMemberAddition(User addedMember,
        User listOwner,UserList list) {
}

@Override
public void onUserListDeletion(User listOwner,
        UserList list) {
}

@Override
public void onUserListCreation(User listOwner,
        UserList list) {
}

@Override
public void onUnfavorite(User source, User target,
```

```
                        Status unfavoritedStatus) {
        }

        @Override
        public void onUnblock(User source, User unblockedUser) {
        }

        @Override
        public void onFriendList(long[] friendIds) {
        }

        @Override
        public void onFollow(User source, User followedUser) {
        }

        @Override
        public void onFavorite(User source, User target,
                        Status favoritedStatus) {
        }

        @Override
        public void onDirectMessage(DirectMessage directMessage) {
        }

        @Override
        public void onDeletionNotice(long directMessageId,
                        long userId) {
        }

        @Override
        public void onBlock(User source, User blockedUser) {
```

```
		}


		@Override
		public void onStallWarning(StallWarning warning) {


		}


	};
}
```

- **Coordenada.java**

```
package com.pfc.issues;


/**
 * Clase para representar una coordenada pura.
 * Esto es un par (x,y) sin información adicional.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class Coordenada {


	/**
	 * Coordenada x (latitud)
	 */
	private double x;


	/**
	 * Coordenada y (longitud)
	 */
	private double y;


	/**
```

```
	 * Constructor de coordenadas
	 * @param x La coordenada x (latitud)
	 * @param y La coordenada y (longitud)
	 */
	public Coordenada(double x,double y) {
		this.x = x;
		this.y = y;
	}

	/**
	 * Construimos una coordenada a partir de una incidencia completa.
	 * @param inc La incidencia completa.
	 */
	public Coordenada(Incidencia inc) {
		this(inc.getX(),inc.getY());
	}

	@Override
	public String toString() {
		return "("+x+","+y+")";
	}

	@Override
	public boolean equals(Object obj) {
		if (obj instanceof Coordenada) {
			Coordenada otra = (Coordenada)obj;
			return equalsCoord(otra);
		}
		else
			return false;
	}
```

```
/**
 * Comprobamos si esta coordenada es igual a otra.
 * Se acepta cierto grado de diferencia, para evitar
 * problemas cuando dos incidencias están demasiado cerca.
 * El grado de similitud necesario es de 5 decimales en ambas
 * coordenadas.
 * @param otra La otra coordenada.
 * @return true si son iguales, false si no.
 */
private boolean equalsCoord(Coordenada otra) {
	return locationCode() == otra.locationCode();
}

/**
 * Generamos un número que simboliza de forma única cada par
 * de coordenadas.
 *
 * Se convierte cada coordenada a un rango positivo y se sesgan
 * sus decimales excepto los 5 primeros. Después se concatenan
 * ambos números.
 *
 * @return El código de las coordenadas.
 */
private long locationCode() {
	StringBuffer sb = new StringBuffer();

	long nX = (int) ((x + 90) * 100000);
	long nY = (int) ((y + 180) * 100000);
	String strX = String.valueOf(nX);
	String strY = String.valueOf(nY);
	sb.append(strX);
	sb.append(strY);
```

```
                long value = Long.parseLong(sb.toString());
                return value;
        }

        @Override
        public int hashCode() {
                long value = locationCode();
                long maxInt = (long)Integer.MAX_VALUE;
                long modulo = value % maxInt;
                return (int)modulo;
        }
}
```

- **Incidencia.java**

```
package com.pfc.issues;


import java.io.IOException;
import java.io.Serializable;
import java.io.StringReader;
import java.util.Date;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;


import org.jdom2.Attribute;
import org.jdom2.DataConversionException;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.JDOMException;
import org.jdom2.input.SAXBuilder;
import org.jdom2.output.Format;
import org.jdom2.output.XMLOutputter;
import org.xml.sax.InputSource;
```

```
import com.pfc.core.CapaTwitter;
import com.pfc.issues.exceptions.DeprecatedIssueException;
import com.pfc.issues.exceptions.IssueException;
import com.pfc.issues.exceptions.MalformedIssueException;
import com.pfc.issues.exceptions.NotAnIssueException;

/**
 * Define una incidencia de tráfico geoposicionada en el mapa.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class Incidencia implements Serializable {

	private static final long serialVersionUID = 5847959217811522363L;

	/**
	 * Mapa que referencia ids de tipo de incidencia con su valor.
	 */
	private static HashMap<Integer,TipoIncidencia> mapa;

	/**
	 * Lista de valores de incidencia que ya no se usan.
	 */
	private static Set<Integer> deprecated;

	static {
		mapa = new HashMap<Integer, TipoIncidencia>();
		deprecated = new HashSet<Integer>();
		for (TipoIncidencia tipo : TipoIncidencia.values()) {
			mapa.put(tipo.getValue(),tipo);
```

```
                if (!tipo.isActivo())
                        deprecated.add(tipo.getValue());
        }
}

/**
 * Id del tweet donde se incluye la incidencia.
 */
private long idTweet;

/**
 * Usuario de Twitter que crea la incidencia
 */
private Usuario usuario;

/**
 * Coordenada x
 */
private double x;

/**
 * Coordenada y
 */
private double y;

/**
 * Timestamp
 */
private long t;

/**
 * Texto de acompañamiento
```

```
     */
    private String observaciones;

    /**
     * Tipo de incidencia
     */
    private TipoIncidencia tipo;

    /**
     * Devolvemos un tipo de incidencia ya existente a partir
     * de su valor.
     *
     * @param value El valor a buscar.
     * @return El tipo de incidencia asignado a dicho valor.
     */
    public static TipoIncidencia getFromValue(int value) {
        return mapa.get(value);
    }

    /**
     * Construimos una incidencia.
     *
     * @param idTweet El id del tweet
     * @param usuario El usuario de twitter
     * @param x La x
     * @param y La y
     * @param t La marca de tiempo.
     * @param observaciones El texto de acompañamiento
     * @param idIncidencia El id del tipo de incidencia
     */
    public Incidencia(long idTweet,Usuario usuario,
            double x,double y,long t,String observaciones,
```

```
                    int idIncidencia) {
            this.idTweet = idTweet;
            this.usuario = usuario;
            this.x = x;
            this.y = y;
            this.t = t;
            this.observaciones = observaciones;
            this.tipo = getFromValue(idIncidencia);
    }

    /**
     * Construimos una incidencia.
     *
     * @param idTweet El id del tweet
     * @param idUser El id de usuario
     * @param userName El nombre de usuario
     * @param x La x
     * @param y La y
     * @param t La marca de tiempo.
     * @param observaciones El texto de acompañamiento
     * @param idIncidencia El id del tipo de incidencia
     */
    public Incidencia(long idTweet,long idUser,String userName,
                    double x,double y,long t,String observaciones,
                    int idIncidencia) {
            this(idTweet,new Usuario(idUser,userName),x,y,t,
                            observaciones,idIncidencia);
    }

    /**
     * Devuelve el id del tweet.
     * @return El id del tweet.
```

```
 */
public long getIdTweet() {
	return idTweet;
}

/**
 * Devuelve el usuario en Twitter.
 * @return El usuario.
 */
public Usuario getUsuario() {
	return usuario;
}

/**
 * Devuelve la coordenada x.
 * @return La coordenada x
 */
public double getX() {
	return x;
}

/**
 * Devuelve la coordenada y.
 * @return La coordenada y.
 */
public double getY() {
	return y;
}

/**
 * Devuelve la marca de tiempo del punto.
 * @return La marca de tiempo.
```

```
	 */
	public long getT() {
		return t;
	}

	/**
	 * Devuelve el texto de acompañamiento.
	 * @return El texto.
	 */
	public String getObservaciones() {
		return observaciones;
	}

	/**
	 * Devuelve el tipo de incidencia.
	 * @return El tipo de incidencia.
	 */
	public TipoIncidencia getTipo() {
		return tipo;
	}

	/**
	 * Obtenemos una incidencia desde xml.
	 *
	 * @param idTweet El id del tweet.
	 * @param idUser El id del usuario.
	 * @param userName El nombre de usuario
	 * @param texto El texto del tweet.
	 *
	 * @return La incidencia parseada.
	 *
	 * @throws IssueException Si ocurre algun error al parsear.
```

```
     */
    public static Incidencia getFromXml(long idTweet,
                long idUser,String userName,String texto)
                throws IssueException {
            String xml = texto.replaceFirst(
                        CapaTwitter.ACCOUNT_NAME + " ","");
            SAXBuilder sb=new SAXBuilder();
            InputSource source = new InputSource(new
StringReader(xml));
            Document doc;
            try {
                doc = sb.build(source);
            } catch (JDOMException e) {
                throw new NotAnIssueException(xml);
            } catch (IOException e) {
                throw new NotAnIssueException(xml);
            }

            Element element=doc.getRootElement();

            if (element.getName().equals("inc")) {
                Incidencia inc = Incidencia.getFromElement(idTweet,
                        idUser,userName,element);
                return inc;
            }
            else
                throw new NotAnIssueException(element.getName());
    }

    /**
     * Obtiene una incidencia a partir de un elemento DOM.
     *
```

```
         * @param idTweet El id del tweet
         * @param idUser El id de usuario
         * @param userName El nombre de usuario
         * @param element El elemento DOM.
         * @return La incidencia construida.
         * @throws IssueException Ocurre si el xml del punto está
         * mal generado.
         */
        private static Incidencia getFromElement(long idTweet,
                        long idUser,String userName,Element element) throws
                IssueException {


                double x,y;
                long t;
                int tipo;
                String texto;


                Attribute xAtt = element.getAttribute("x");
        //OBL
                Attribute yAtt = element.getAttribute("y");
        //OBL
                Attribute tAtt = element.getAttribute("t");
        //OPC
                Attribute textAtt = element.getAttribute("text");          //OPC
                Attribute tipoAtt = element.getAttribute("i");             //OBL
                if (xAtt==null)
                        throw new
MalformedIssueException(element.getName(),
                                        "Falta el atributo x");
                if (yAtt==null)
                        throw new
MalformedIssueException(element.getName(),
                                        "Falta el atributo y");
                if (tipoAtt==null)
```

```
                throw new
MalformedIssueException(element.getName(),
                                "Falta el atributo i");
        try {
                x = xAtt.getDoubleValue();
        } catch (DataConversionException e) {
                throw new
MalformedIssueException(element.getName(),
                                "El atributo x no tiene formato
correcto");
        }
        try {
                y = yAtt.getDoubleValue();
        } catch (DataConversionException e) {
                throw new
MalformedIssueException(element.getName(),
                                "El atributo y no tiene formato
correcto");
        }
        try {
                tipo = tipoAtt.getIntValue();
                if (deprecated.contains(tipo)) {
                        TipoIncidencia ti =
Incidencia.getFromValue(tipo);
                        throw new DeprecatedIssueException(ti,
                                        "Incidencia deprecated");
                }
        } catch (DataConversionException e) {
                throw new
MalformedIssueException(element.getName(),
                                "El atributo i no tiene formato
correcto");
        }
        if (tAtt==null) {
                t = new Date().getTime();
```

```
		}
		else {
			try {
				t = tAtt.getLongValue();
			} catch (DataConversionException e) {
				t = new Date().getTime();
			}
		}
		if (textAtt==null) {
			texto = "";
		}
		else {
			texto = textAtt.getValue();
		}

		return new Incidencia(idTweet,idUser,userName,
				x, y, t, texto,tipo);

	}

	/**
	 * Escribe el xml correspondiente a la incidencia.
	 */
	public String serialize() {
		Document d = serializeDoc();

		Format format=Format.getCompactFormat();
	format.setOmitEncoding(true);
	format.setOmitDeclaration(true);
	XMLOutputter outputter=new XMLOutputter(format);
	String xml=outputter.outputString(d);
		return xml;
```

```
}

/**
 * Serializamos la incidencia como Documento DOM (en XML).
 *
 * @return La incidencia serializada.
 */
private Document serializeDoc() {
	Element element=new Element("inc");
element.setAttribute("x",String.valueOf(x));
element.setAttribute("y",String.valueOf(y));
element.setAttribute("t",String.valueOf(t));
element.setAttribute("id_tweet",String.valueOf(idTweet));
element.setAttribute("id_user",String.valueOf(
		usuario.getIdUsuario()));
if (observaciones!=null && !observaciones.isEmpty())
	element.setAttribute("text",observaciones);
element.setAttribute("i",String.valueOf(tipo.getValue()));
Document document=new Document(element);
return document;
}

/**
 * Escribimos la incidencia en el formato CSV usado para
 * comunicarnos mediante AJAX.
 *
 * Se escapan aquellos caracteres que pueden ser conflictivos.
 *
 * @return La cadena con el CSV ya construida.
 */
public String ajaxPrint() {
	StringBuffer buff = new StringBuffer();
```

```
		buff.append(tipo.getValue());
		buff.append(";");
		buff.append(x);
		buff.append(";");
		buff.append(y);
		buff.append(";");
		String textoEsc = Incidencia.escaparCSV(observaciones);
		buff.append(textoEsc);
		buff.append(";");
		buff.append(usuario.getUserName());
		buff.append(";");
		buff.append(t);

		return buff.toString();
	}

	public boolean equals(Object o) {
		if (o instanceof Incidencia)
			return equalsIssue((Incidencia)o);
		else
			return false;
	}

	/**
	 * Comprobamos si dos incidencias son iguales.
	 * @param i La otra incidencia.
	 * @return true si son iguales, false si no.
	 */
	public boolean equalsIssue(Incidencia i) {
		return x==i.getX() && y==i.getY() && t==i.getT()
				&& tipo==i.getTipo();
```

```
	}

	@Override
	public String toString() {
		return serialize();
	}

	/**
	 * Escapamos una cadena que forma parte de un formato CSV.
	 * Para ello escapamos los \n y los ;
	 * @param texto El texto a escapar.
	 * @return El texto escapado.
	 */
	public static String escaparCSV(String texto) {
		return texto.replaceAll("\\n", "[LINEFEED]").
					replaceAll(";","[SEMICOLON]");
	}
}
```

- **TipoIncidencia.java**

```
package com.pfc.issues;

/**
 * Tipo enumerado para determinar los tipos de incidencia permitidos.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public enum TipoIncidencia {
	ACCIDENTE(0,true,"Accidente"),
	VEHICULO_ABANDONADO(1,true,"Vehículo abandonado"),
	CONGESTION_TRAFICO(2,true,"Congestión de tráfico"),
	VEHICULO_MAL_ESTACIONADO(3,true,"Vehículo mal estacionado"),
```

```
ESTACIONAMIENTO_LIBRE(4,true,"Estacionamiento libre"),
PUNTO_NEGRO(5,false,"Punto negro"),
RADAR(6,false,"Radar"),
OBRAS(7,true,"Obras"),
INCIDENCIA_METEOROLOGICA(8,true,"Incidencia meteorológica"),
CALLE_CORTADA(9,true,"Calle cortada"),
EVENTOS(10,true,"Eventos"),
OTROS(11,true,"Otros");

/**
 * El valor que representa el tipo de incidencia.
 */
private int value;

/**
 * Define si el valor está activo o desechado.
 */
private boolean activo;

/**
 * EL texto que representa el tipo de incidencia.
 */
private String texto;

/**
 * Devuelve el valor del tipo de incidencia
 * @return EL valor del tipo de incidencia
 */
public int getValue() {
        return value;
}
```

```
/**
 * Determina si el valor está activo o no.
 * @return Si el valor está activo o no.
 */
public boolean isActivo() {
        return activo;
}

/**
 * Devuelve el texto de la incidencia.
 * @return El texto.
 */
public String getTexto() {
        return texto;
}

/**
 * Creamos un nuevo tipo de incidencia con su valor correspondiente.
 * @param value El valor a asignar al tipo.
 * @param activo Si el valor está activo o no.
 * @param texto El texto a mostrar.
 */
private TipoIncidencia(int value,boolean activo,String texto) {
        this.value = value;
        this.activo = activo;
        this.texto = texto;
}

/**
 * Devuelve la representación del tipo de incidencia como
 * icono para el buscador de la web.
 * @return La representación HTML del tipo de incidencia.
```

```
	 */
	public String webPrint() {
		StringBuffer buffer = new StringBuffer();


		buffer.append("<li><a id='icon");
		buffer.append(value);
		buffer.append("' href='javascript:swapIcon(");
		buffer.append(value);
		buffer.append(")'><img id='img_icon");
		buffer.append(value);
		buffer.append("'src='iconos/icono");
		buffer.append(value);
		buffer.append(".png' width='20' height='20' alt='");
		buffer.append(texto);
		buffer.append("' title='");
		buffer.append(texto);
		buffer.append("' /></a></li>");


		return buffer.toString();
	}
}
```

- **Usuario.java**

```
package com.pfc.issues;


/**
 * Representación de un usuario de Twitter.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class Usuario {

```

```
/**
 * Id del usuario en Twitter.
 */
private long idUsuario;

/**
 * Nombre de usuario en Twitter.
 */
private String userName;

/**
 * Creamos un usuario, conociendo su nombre de usuario.
 * @param idUsuario Id del usuario.
 * @param userName Nombre del usuario.
 */
public Usuario(long idUsuario, String userName) {
	this.idUsuario = idUsuario;
	this.userName = userName;
}

/**
 * Creamos un usuario sin nombre.
 * @param idUsuario El id del usuario.
 */
public Usuario(long idUsuario) {
	this.idUsuario = idUsuario;
	this.userName = null;
}

/**
 * Devuelve el id del usuario.
 * @return El id del usuario.
```

```
	 */
	public long getIdUsuario() {
		return idUsuario;
	}

	/**
	 * Devuelve el nombre del usuario.
	 * @return El nombre del usuario.
	 */
	public String getUserName() {
		return userName;
	}


}
```

- **BD.java**

```
package com.pfc.issues.dao;


import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;


import com.pfc.issues.Incidencia;
import com.pfc.issues.Usuario;
```

```
/**
 * Clase para gestionar todas las conexiones a la BD.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class BD {

	/**
	 * Instancia de conexión a la BD.
	 */
	private static BD instance;

	/**
	 * Representa el rango de tiempo HOY.
	 */
	private static final int HOY = 1;
	/**
	 * Representa el rango de tiempo SEMANA.
	 */
	private static final int SEMANA = 2;
	/**
	 * Representa el rango de tiempo MES.
	 */
	private static final int MES = 3;
	/**
	 * Representa el rango de tiempo SIEMPRE.
	 */
	private static final int SIEMPRE = 4;

	/**
	 * Número de milisegundos para un día.
```

```
 */
private static final long MS_HOY=1000*60*60*24;

/**
 * Número de milisegundos para una semana.
 */
private static final long MS_SEMANA = MS_HOY * 7;

/**
 * Número de milisegundos para un mes.
 */
private static final long MS_MES = MS_HOY * 30;

/**
 * Conexión interna a la BD (según JDBC).
 */
private Connection conexion;

/**
 * Asignamos ids de usuario a su usuario.
 * Sirve para optimizar asignaciones, teniendo que realizar
 * una única consulta.
 */
private HashMap<Long,Usuario> mapaUsuariosId;

/**
 * Asignamos nombres de usuario a su usuario.
 * Sirve para optimizar asignaciones, teniendo que realizar
 * una única consulta.
 */
private HashMap<String,Usuario> mapaUsuariosNombre;
```

```
/**
 * Conseguimos la instancia Singleton del acceso a la BD.
 * @return Objeto de acceso a la BD.
 */
public static BD getInstance() {
		if (instance == null)
			instance = new BD();
		return instance;
}

/**
 * Constructor privado de acceso a la BD.
 */
private BD() {
		try {
			Class.forName("com.mysql.jdbc.Driver");
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}
		try {
			conexion = DriverManager.getConnection(
					"jdbc:mysql://localhost/pfc","root","");
		} catch (SQLException e) {
			e.printStackTrace();
		}
		mapaUsuariosId = new HashMap<Long, Usuario>();
		mapaUsuariosNombre = new HashMap<String, Usuario>();
}

/**
 * Devolvemos un usuario a partir de su id.
 * Se busca en caché, y si no en BD.
```

```
 * @param id El id del usuario.
 * @return El usuario.
 */
public Usuario getUsuario(long id) {
	Usuario user = mapaUsuariosId.get(id);
	if (user==null) {
		user = buscarUsuario(id);
		if (user != null) {
			mapaUsuariosId.put(user.getIdUsuario(),user);

mapaUsuariosNombre.put(user.getUserName(),user);
		}
	}
	return user;
}

/**
 * Devolvemos un usuario a partir de su nombre.
 * Se busca en caché, y si no en BD.
 * @param nombre El nombre del usuario.
 * @return El usuario.
 */
public Usuario getUsuario(String nombre) {
	Usuario user = mapaUsuariosNombre.get(nombre);
	if (user==null) {
		user = buscarUsuario(nombre);
		if (user != null) {
			mapaUsuariosId.put(user.getIdUsuario(),user);

mapaUsuariosNombre.put(user.getUserName(),user);
		}
	}
	return user;
```

```
	}


	/**
	 * Devolvemos un usuario a partir de su id.
	 * Se busca en BD.
	 * @param id El id del usuario.
	 * @return El usuario.
	 */
	private Usuario buscarUsuario(long id) {
		try {
			PreparedStatement ps = conexion.prepareStatement(
					"SELECT * FROM usuarios WHERE id =
?");
			ps.setLong(1,id);

			ResultSet rs = ps.executeQuery();

			while (rs.next()) {
				String userName = rs.getString("nombre");
				return new Usuario(id,userName);
			}

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

		return null;
	}

	/**
	 * Devolvemos un usuario a partir de su nombre.
	 * Se busca en BD.
```

```
	 * @param nombre El nombre del usuario.
	 * @return El usuario.
	 */
	private Usuario buscarUsuario(String nombre) {
		try {
			PreparedStatement ps = conexion.prepareStatement(
					"SELECT * FROM usuarios WHERE
nombre = ?");
			ps.setString(1,nombre);

			ResultSet rs = ps.executeQuery();

			while (rs.next()) {
				Long id = rs.getLong("id");
				return new Usuario(id,nombre);
			}

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

		return null;
	}

	/**
	 * Devuelve una lista con todas las incidencias de la BD.
	 *
	 * @return Lista de incidencias de la BD.
	 */
	public List<Incidencia> getIncidencias() {
		List<Incidencia> incidencias = new ArrayList<Incidencia>();
```

```
		try {
			Statement s = conexion.createStatement();
			ResultSet rs = s.executeQuery ("SELECT * FROM
incidencias");
			while(rs.next()) {
				long idTweet = rs.getLong("id_tweet");
				long idUser = rs.getLong("id_user");
				int tipo = rs.getInt("tipo");
				String observaciones =
rs.getString("observaciones");
				long t = rs.getLong("timestamp");
				double x = rs.getDouble("x");
				double y = rs.getDouble("y");

				Incidencia inc = new Incidencia(idTweet,
						getUsuario(idUser),
						x, y, t, observaciones,tipo);
				incidencias.add(inc);
			}
		} catch (SQLException e) {
			e.printStackTrace();
		}
		return incidencias;
	}

	/**
	 * Devolvemos una serie de incidencias de la BD.
	 * @param where La clausula WHERE de la consulta SQL.
	 * @return Las incidencias que cumplan la condición.
	 */
	private List<Incidencia> getIncidenciasWhere(String where) {
		List<Incidencia> incidencias = new ArrayList<Incidencia>();
```

```
                try {
                        String consulta = "SELECT * FROM incidencias WHERE
";
                        consulta += where;
                        Statement s = conexion.createStatement();
                        ResultSet rs = s.executeQuery(consulta);
                        while (rs.next()) {
                                long idTweet = rs.getLong("id_tweet");
                                long idUser = rs.getLong("id_user");
                                int tipo = rs.getInt("tipo");
                                String observaciones =
rs.getString("observaciones");
                                long t = rs.getLong("timestamp");
                                double x = rs.getDouble("x");
                                double y = rs.getDouble("y");

                                Incidencia inc = new Incidencia(idTweet,
                                                getUsuario(idUser), x,
                                                y, t, observaciones, tipo);
                                incidencias.add(inc);
                        }
                } catch (SQLException e) {
                        e.printStackTrace();
                }
                return incidencias;
        }

        /**
         * Devolvemos las incidencias filtradas por varios criterios
         * (opcionales)
         *
         * @param tipos La lista de tipos de incidencias a permitir.
         * @param nombreUsuario El nombre del usuario a filtrar.
```

```
 * @param idTiempo El identificador del rango de tiempo
 * @param lastTime El timestamp a partir del cual filtrar.
 *
 * @return La lista de incidencias que cumplen los criterios.
 */
public List<Incidencia> getIncidencias(List<Integer> tipos,
			String nombreUsuario,Long idTiempo,Long lastTime) {
		StringBuffer where = new StringBuffer();
		boolean primero = true;
		if (tipos!=null && !tipos.isEmpty()) {
			if (!primero) {
				where.append(" AND ");
			}
			else {
				primero = false;
			}
			StringBuffer bufferIn = new StringBuffer();
			bufferIn.append("tipo IN (");
			for (int i=0;i<tipos.size()-1;i++) {
				bufferIn.append(tipos.get(i));
				bufferIn.append(",");
			}
			bufferIn.append(tipos.get(tipos.size()-1));
			bufferIn.append(")");
			where.append(bufferIn);
		}

		Long idUsuario = null;
		if (nombreUsuario != null && !nombreUsuario.isEmpty()) {
			Usuario usuario = getUsuario(nombreUsuario);
			if (usuario!=null)
				idUsuario = usuario.getIdUsuario();
```

```
            else
                return new ArrayList<Incidencia>();
        }
        if (idUsuario!=null) {
            if (!primero) {
                where.append(" AND ");
            }
            else {
                primero = false;
            }
            where.append("id_user = " + idUsuario + " ");
        }

        Long tiempoAtras = BD.getMsFromId(idTiempo);
        if (tiempoAtras!=null) {
            if (!primero) {
                where.append(" AND ");
            }
            else {
                primero = false;
            }
            long tiempoMinimo = new Date().getTime() -
tiempoAtras;
            where.append("timestamp >= " + tiempoMinimo + " ");
        }

        if (lastTime!=null) {
            if (!primero) {
                where.append(" AND ");
            }
            else {
                primero = false;
```

```
                }
                where.append("timestamp > " + lastTime + " ");
        }

        String whereStr = where.toString();
        if (whereStr.isEmpty()) {
                return getIncidencias();
        }
        else {
                return getIncidenciasWhere(whereStr);
        }
}

/**
 * Devolvemos la ultima incidencia en la BD.
 *
 * @return La última incidencia del BD.
 */
public Incidencia getLastIncidencia() {

        String consulta = "SELECT * FROM incidencias ";
        consulta += "WHERE timestamp = ( ";
        consulta += "SELECT MAX( timestamp )";
        consulta += "FROM incidencias )";

        try {
                Statement s = conexion.createStatement();
                ResultSet rs = s.executeQuery(consulta);

                while(rs.next()) {
                        long idTweet = rs.getLong("id_tweet");
                        long idUser = rs.getLong("id_user");
```

```
                                        int tipo = rs.getInt("tipo");
                                        String observaciones =
rs.getString("observaciones");
                                        long t = rs.getLong("timestamp");
                                        double x = rs.getDouble("x");
                                        double y = rs.getDouble("y");

                                        Incidencia inc = new
Incidencia(idTweet,getUsuario(idUser),
                                                        x, y, t, observaciones,tipo);
                                        return inc;
                                }

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

                return null;

        }

        /**
         * Insertamos una incidencia a la BD.
         *
         * @param inc La incidencia a insertar.
         */
        public void addIncidencia(Incidencia inc) {

                long idUser = inc.getUsuario().getIdUsuario();
                if (!mapaUsuariosId.containsKey(idUser) &&
                                buscarUsuario(idUser) == null) {
                        addUsuario(inc.getUsuario());
                        mapaUsuariosId.put(idUser,inc.getUsuario());
```

```
		}
		try {
			PreparedStatement ps = conexion.prepareStatement(
					"INSERT INTO incidencias
VALUES(?,?,?,?,?,?,?)");
			ps.setLong(1,inc.getIdTweet());
			ps.setLong(2,inc.getUsuario().getIdUsuario());
			ps.setInt(3,inc.getTipo().getValue());
			ps.setString(4,inc.getObservaciones());
			ps.setLong(5,inc.getT());
			ps.setDouble(6,inc.getX());
			ps.setDouble(7,inc.getY());

			ps.execute();
		} catch (SQLException e) {
			e.printStackTrace();
		}

	}

	/**
	 * Insertamos un usuario en la BD.
	 * @param user El usuario a insertar.
	 */
	public void addUsuario(Usuario user) {
		try {
			PreparedStatement ps = conexion.prepareStatement(
					"INSERT INTO usuarios VALUES(?,?)");
			ps.setLong(1,user.getIdUsuario());
			ps.setString(2,user.getUserName());
			ps.executeUpdate();
		} catch (SQLException e) {
```

```
                        e.printStackTrace();
                }
        }

        /**
         * Insertamos una lista de incidencias a la BD.
         *
         * @param incidencias La lista de incidencias.
         */
        public void addIncidencias(List<Incidencia> incidencias) {
                if (incidencias!=null) {
                        for (Incidencia inc : incidencias)
                                addIncidencia(inc);
                }
        }

        /**
         * Insertamos aquellas incidencias de la lista que todavía no
         * se encuentran en la BD.
         *
         * @param incidencias Las incidencias a insertar.
         */
        public void addNewIncidencias(List<Incidencia> incidencias) {
                Incidencia ultima = getLastIncidencia();
                if (ultima==null)
                        addIncidencias(incidencias);
                else {
                        for (Incidencia inc : incidencias) {
                                if (inc.getT() > ultima.getT())
                                        addIncidencia(inc);
                        }
                }
```

```
}

/**
 * Borramos todas las incidencias de la BD.
 */
public void deleteIncidencias() {
	try {
		Statement s = conexion.createStatement();
		s.executeUpdate("DELETE FROM incidencias");
	} catch (SQLException e) {
		e.printStackTrace();
	}
}

/**
 * Devuelve el número de milisegundos de un rango de tiempo
 * dado por su id.
 * @param idTiempo El id del rango de tiempo.
 * @return Los milisegundos que corresponden.
 */
private static Long getMsFromId(Long idTiempo) {
	if (idTiempo == null) {
		return null;
	}
	else {
		switch(idTiempo.intValue()) {
		case HOY :
			return MS_HOY;
		case SEMANA :
			return MS_SEMANA;
		case MES :
			return MS_MES;
```

```
                        case SIEMPRE :
                                return null;
                        }
                        return null;
                }
        }
}
```

- **DeprecatedIssueException.java**

```
package com.pfc.issues.exceptions;

import com.pfc.issues.TipoIncidencia;

/**
 * Excepción para controlar aquellas incidencias que ya no se encuentran
 * activas por el sistema.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class DeprecatedIssueException extends IssueException {

        private static final long serialVersionUID = 8442726146967060163L;

        /**
         * El tipo de incidencia que está desechada.
         */
        private TipoIncidencia tipo;

        /**
         * El mensaje a mostrar.
         */
        private String msg;
```

```
	/**
	 * Creamos una excepción del tipo deprecated.
	 * @param tipo El tipo de incidencia ya desechado.
	 * @param msg El mensaje a mostrar.
	 */
	public DeprecatedIssueException(TipoIncidencia tipo,String msg) {
		super(msg);

		this.tipo = tipo;
		this.msg = msg;
	}

	@Override
	public String getMessage() {
		return msg;
	}

	/**
	 * Devuelve el tipo de incidencia.
	 * @return El tipo de incidencia.
	 */
	public TipoIncidencia getTipo() {
		return tipo;
	}

}
```

- **IssueException.java**

```
package com.pfc.issues.exceptions;


/**
 * Clase para encapsular las posibles excepciones que se pueden dar.
```

```
 * Sólo se lanzara sin particularizar en caso de que el error
 * sea indeterminado.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class IssueException extends Exception {

	private static final long serialVersionUID = 5212868799770959492L;

	/**
	 * Construimos un error de entidad con un mensaje de error.
	 * @param s El mensaje de error.
	 */
	public IssueException(String s) {
		super(s);
	}

	@Override
	public String getMessage() {
		return "Excepcion desconocida al leer incidencia";
	}
}
```

- **MalformedIssueException.java**

```
package com.pfc.issues.exceptions;

/**
 * Definimos una excepción de entidad para el caso de que una
 * entidad tenga un formato incorrecto.
 *
 * @author Alejandro Pérez Manzano.
 *
```

```
 */
public class MalformedIssueException extends IssueException {

	private static final long serialVersionUID = -3931916498812876430L;

	private String elemento;
	private String msg;

	/**
	 * Construimos el error de un elemento de una entidad.
	 * @param nombreElemento El nombre del elemento.
	 * @param msg El mensaje concreto de error.
	 */
	public MalformedIssueException(String nombreElemento,String msg) {
		super(msg);
		this.elemento = nombreElemento;
		this.msg = msg;
	}

	@Override
	public String getMessage() {
		String s = "Error en el elemento : " + elemento;
		s += "\n " + msg;
		return s;
	}

}
```

- **NotAnIssueException.java**

```
package com.pfc.issues.exceptions;

/**
 * Definimos una excepción en el caso de encontrar un xml que
```

```
 * no incluya una incidencia como contenido.
 *
 * @author Alejandro Pérez Manzano.
 *
 */
public class NotAnIssueException extends IssueException {


	private static final long serialVersionUID = 4918101705244434614L;


	/**
	 * Creamos un error de entidad desconocida.
	 * @param nombreElemento El nombre del elemento.
	 */
	public NotAnIssueException(String s) {
		super(s);
	}


	public String getMessage() {
		return "No se encuentra incidencia en el tweet";
	}


}
```

- **Servidor.java**

```
package com.pfc.server;


import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;


import com.pfc.core.CapaTwitter;
import com.pfc.issues.Coordenada;
```

```
import com.pfc.issues.Incidencia;
import com.pfc.issues.dao.BD;

/**
 * Crea el servidor que gestiona la entrada de incidencias a través de
 * Twitter y las peticiones de incidencias desde el front-end web
 * a través de AJAX.
 *
 * @author Alejandro Pérez Manzano
 *
 */
public class Servidor {

	/**
	 * Creamos una instancia singleton del servidor.
	 */
	private static Servidor instance;

	/**
	 * El objeto de la capa de Twitter.
	 */
	private CapaTwitter capa;

	/**
	 * Crea un nuevo servidor.
	 */
	private Servidor() {
		capa=new CapaTwitter();
	}

	/**
	 * Devolvemos una instancia Singleton del servidor.
```

```
	 * @return Devolvemos un Singleton Servidor.
	 */
	public static Servidor getInstance() {
		if (instance==null) {
			instance = new Servidor();
		}
		return instance;
	}

	/**
	 * Carga los nuevos tweets recibidos a esta cuenta.
	 */
	public void cargarNuevos() {
		capa.cargarNuevos();
	}

	/**
	 * Realizamos una búsqueda filtrada y devolvemos sus incidencias
	 * procesadas.
	 *
	 * @param tipos Los tipos de incidencia aceptados.
	 * @param nombreUsuario El nombre de usuario a buscar.
	 * @param idTiempo El id del rango de tiempo.
	 * @param lastTime El timestamp a partir del cual filtrar.
	 *
	 * @return Las incidencias filtradas y procesadas.
	 */
	public Map<Coordenada,Incidencia> getIncidencias(
			List<Integer> tipos,String nombreUsuario,Long
idTiempo,
			Long lastTime) {
		BD bd = BD.getInstance();
```

```
		List<Incidencia> incidencias = bd.getIncidencias(tipos,
				nombreUsuario, idTiempo,lastTime);
		if (incidencias == null)
			return null;
		else {
			Map<Coordenada,Incidencia> puntos =
					new HashMap<Coordenada,
Incidencia>();
			for (Incidencia inc : incidencias) {
				Coordenada c = new Coordenada(inc);
				puntos.put(c,inc);
			}
			return puntos;
		}
	}
	/**
	 * Devolvemos la lista de tipos codificada como String
	 * en una petición de búsqueda.
	 * @param tipos Los tipos como String.
	 * @return La lista de tipos de incidencias.
	 */
	public static List<Integer> procesarTipos(String tipos) {
		if (tipos == null || tipos.isEmpty())
			return null;
		else {
			List<Integer> listaTipos = new ArrayList<Integer>();
			String[] trozos = tipos.split(",");
			if (trozos != null) {
				for (String trozo : trozos) {
					try {
						int tipo =
Integer.parseInt(trozo);
						listaTipos.add(tipo);
```

```
                                                } catch (NumberFormatException e) {
                                                }
                                        }
                                }
                                return listaTipos;
                        }
                }

}
```

- **index.jsp**

```
<%@page import="com.pfc.issues.Incidencia"%>
<%@page import="com.pfc.issues.TipoIncidencia"%>
<%@page import="java.util.HashMap"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
<link rel="stylesheet" href="css/index.css" type="text/css">
<script type="text/javascript"
        src="https://maps.google.com/maps/api/js?sensor=true">
</script>
<script type="text/javascript" src="js/ajax.js">
</script>
<script type="text/javascript" src="js/gmaps.js">
</script>

</head>
<body onload="initialize()">
```

```
	<div id="search_bar">
		<ul>
			<%
				TipoIncidencia[] incidencias =
TipoIncidencia.values();
				for (TipoIncidencia inc : incidencias) {
					if (inc.isActivo()) {
						out.write(inc.webPrint());
					}
				}
			%>
			<li>
			<a id="iconActivar" href="javascript:activarIconos()">
			<img id="img_iconActivar" src="iconos/activar.png"
				width="25" height="25"
				alt="Activar todos" title="Activar todos" />
			</a>
			</li>
			<li>
			<a id="iconDesactivar"
href="javascript:desactivarIconos()">
			<img id="img_iconDesactivar"
src="iconos/desactivar.png"
				width="20" height="20"
				alt="Desactivar todos" title="Desactivar todos"
/>
			</a>
			</li>
			<li class="primero">
				<label for="usuario">Usuario:</label>
			</li>
			<li class="chico"><input id="usuario" type="text"
/></li>
			<li class="primero">
```

```
                    <label for="tiempo">Tiempo:</label>
                </li>
                <li class="chico"><select id="tiempo" >
                    <option value="1">Hoy</option>
                    <option value="2">Esta semana</option>
                    <option value="3">Este mes</option>
                    <option value="4">Siempre</option>
                </select></li>
                <li class="primero"><a id="botonBuscar"
                    href="javascript:buscar()"
class="boton">Buscar</a></li>
                <li><a id="botonCentrar"
href="javascript:centrarMapa()"
                    class="boton">Centrar</a></li>
            </ul>
        </div>
        <div id="map_canvas" ></div>
</body>
</html>
```

- **search.jsp**

```
<%@page import="java.util.ArrayList"%>
<%@page import="com.pfc.issues.Coordenada"%>
<%@page import="java.util.HashMap"%>
<%@page import="java.util.Map"%>
<%@page import="com.pfc.issues.Incidencia"%>
<%@page import="java.util.List"%>
<%@page import="com.pfc.server.Servidor"%>
<%@page import="twitter4j.*"%>
<%@ page contentType="text/html; charset=iso-8859-1" language="java"%>
<%
        List<Integer> tipos = null;
        String nombreUsuario = null;
```

```
Long tiempo = null;
Long lastTime = null;

String paramTipos = request.getParameter("tipos");
String paramUsuario = request.getParameter("usuario");
String paramTiempo = request.getParameter("tiempo");
String paramLast = request.getParameter("last");

if (paramTipos!=null && !paramTipos.isEmpty()) {
        tipos = Servidor.procesarTipos(paramTipos);
}

if (paramUsuario!=null && !paramUsuario.isEmpty()) {
        nombreUsuario = paramUsuario;
}

if (paramTiempo!=null && !paramTiempo.isEmpty()) {
        tiempo = Long.parseLong(paramTiempo);
}
if (paramLast!=null && !paramLast.isEmpty()) {
        lastTime = Long.parseLong(paramLast);
}

Servidor server=Servidor.getInstance();

Map<Coordenada,Incidencia> puntos = server.
                getIncidencias(tipos,nombreUsuario,tiempo,lastTime);

if (puntos!=null) {
        for (Incidencia p : puntos.values()) {
                out.println(p.ajaxPrint());
        }
```

```
	}
%>
```

- **ajax.js**

```
/*
 * COMUNICACIÓN AJAX
 */

/**
 * Objeto que gestiona las comunicaciones necesarias para AJAX.
 */
var xmlhttp;

/**
 * El timestamp del mensaje más nuevo.
 */
var lastTime = -1;

/**
 * Obtiene el objeto que procesa las peticiones para AJAX.
 * @returns El objeto de procesamiento de AJAX.
 */
function GetXmlHttpObject() {
	if (window.XMLHttpRequest) {
		return new XMLHttpRequest();
	}
	if (window.ActiveXObject) {
		return new ActiveXObject("Microsoft.XMLHTTP");
	}
	return null;
}

/*
```

```
 * FUNCIONES A SER LLAMADAS POR UI
 */


/**
 * Realizamos una petición de búsqueda en AJAX filtrada
 * según los siguientes parámetros.
 *
 * @param tiposSeleccionados Los tipos de incidencia seleccionados.
 * @param usuario El nombre de usuario a buscar.
 * @param tiempo La clave para los distintos rangos temporales.
 */
function loadSearch(tiposSeleccionados,usuario,tiempo) {
	xmlhttp = GetXmlHttpObject();

	if (xmlhttp == null) {
		alert("Your browser does not support Ajax HTTP");
		return;
	}

	var tiposStr = getStringTipos(tiposSeleccionados);

	var first = true;

	lastTime = -1;

	var url = "search.jsp";
	if (tiposStr != null && tiposStr != '') {
		url += (first ? "?" : "&");
		first = false;
		url += "tipos=" + tiposStr;
	}
	if (usuario != null && usuario != '') {
```

```
                url += (first ? "?" : "&");
                first = false;
                url += "usuario=" + usuario;
        }
        if (tiempo != null && tiempo != '') {
                url += (first ? "?" : "&");
                first = false;
                url += "tiempo=" + tiempo;
        }

        xmlhttp.onreadystatechange = getOutput;
        xmlhttp.open("GET", url, true);
        xmlhttp.send(null);
}

/**
 * Realizamos un polling en AJAX filtrada
 * según los siguientes parámetros.
 *
 * @param tiposSeleccionados Los tipos de incidencia seleccionados.
 * @param usuario El nombre de usuario a buscar.
 * @param tiempo La clave para los distintos rangos temporales.
 */
function loadNew(tiposSeleccionados,usuario,tiempo) {
        xmlhttp = GetXmlHttpObject();

        if (xmlhttp == null) {
                alert("Your browser does not support Ajax HTTP");
                return;
        }

        var tiposStr = getStringTipos(tiposSeleccionados);
```

```
	var first = true;

	var url = "search.jsp";
	if (lastTime != null && lastTime != -1) {
		url += (first ? "?" : "&");
		first = false;
		url += "last=" + lastTime;
	}
	if (tiposStr != null && tiposStr != '') {
		url += (first ? "?" : "&");
		first = false;
		url += "tipos=" + tiposStr;
	}
	if (usuario != null && usuario != '') {
		url += (first ? "?" : "&");
		first = false;
		url += "usuario=" + usuario;
	}
	if (tiempo != null && tiempo != '') {
		url += (first ? "?" : "&");
		first = false;
		url += "tiempo=" + tiempo;
	}

	xmlhttp.onreadystatechange = getOutput;
	xmlhttp.open("GET", url, true);
	xmlhttp.send(null);
}

/*
 * CALLBACKS
```

```
 */

/**
 * Devuelve el resultado de la llamada.
 */
function getOutput() {
        if (xmlhttp.readyState == 4) {
                var txt = xmlhttp.responseText;
                procesarTexto(txt);
                polling();
        }
}

/*
 * FUNCIONES DE PROCESAMIENTO
 */

/**
 * Procesamos el texto devuelto por el servidor jsp que procesa el AJAX.
 * Usa un formato csv para especificar entidades.
 * @param txt La lista de elementos, como texto.
 */
function procesarTexto(txt) {
        txt = txt.trim();
        var lineas = txt.split("\n");
        for (var i=0;i<lineas.length;i++) {

                var linea = lineas[i];
                if (linea.length==0)
                        return;

                var lineaTrozos = linea.split(";");
```

```
                procesarIncidencia(lineaTrozos);
        }
}

/**
 * Leemos un punto en formato csv.
 * @param linea La linea cortada en trozos
 * @returns {google.maps.LatLng} La coordenada del punto en
 * formato de Google Maps.
 */
function readPoint(linea) {
        var x = parseFloat(linea[1]);
        var y = parseFloat(linea[2]);
        return new google.maps.LatLng(x,y);
}

/**
 * Procesamos un punto de Google Maps.
 * Fijamos un marcador en el mapa para posicionar el punto.
 * @param linea La coordenadas del punto, como array de texto.
 */
function procesarIncidencia(linea) {
        var coordGoogle = readPoint(linea);

        var tipoIncidencia = linea[0];
        var indice = parseInt(tipoIncidencia);
        var observaciones = linea[3];
        observaciones = observaciones.replace(/\[LINEFEED\]/g,"\n");
        observaciones = observaciones.replace(/\[SEMICOLON\]/g,";");
        var userName = linea[4];
        var t = linea[5];
        var ms = parseInt(t);
```

```
        if (ms > lastTime) {
                lastTime = ms;
        }
        var date = new Date(ms);

        var text = "";
        text += getNombreIncidencia(indice) + "\n";
        text += "Usuario : @" + userName + "\n";
        text += "Dia : " + date.toLocaleDateString() + "\n";
        text += "Hora : " + date.toLocaleTimeString() + "\n";
        text += "Observaciones : " + observaciones;

        var icono = getIcon(tipoIncidencia);
        var marker = new google.maps.Marker({
           position: coordGoogle,
           title : text,
           icon : icono
        });

        addMarker(marker);
}

/**
 * Devuelve el icono del tipo de incidencia, escalado a tamaño.
 * @param indice El indice de la incidencia (puede ser erroneo)
 * @returns El icono de la incidencia
 */
function getIcon(indice) {
        var urlIcon = "iconos/";
        if (indice == NaN || indice < 0 || indice > incidencias.length) {
                urlIcon += "default.png";
                size = new google.maps.Size(25,25);
```

```
        }
        else {
                urlIcon += "icono" + indice + ".png";
                size = new google.maps.Size(
                                incidencias[indice][1],incidencias[indice][2]);
        }

        var icon = {
                url : urlIcon,
                scaledSize : size
        };

        return icon;
}

/**
 * Devuelve el nombre de la incidencia en base a su indice.
 * @param indice El indice de la incidencia (puede ser erroneo).
 * @returns El nombre de la incidencia.
 */
function getNombreIncidencia(indice) {
        if (indice == NaN || indice < 0 || indice >= incidencias.length) {
                return 'Error';
        }
        else {
                return incidencias[indice][0];
        }
}

/**
 * Devuelve un string (separado por comas) que representa a todos
 * los tipos de incidencia a buscar, para ser enviado por url.
```

```
 *
 * @param tipos La lista de tipos de incidencia
 *
 * @returns El string codificado para la URL.
 */
function getStringTipos(tipos) {
	if (tipos == null || tipos.length == 0) {
		return null;
	}
	var str="";
	for (var i=0;i<tipos.length-1;i++) {
		str+=tipos[i]+",";
	}
	str+=tipos[tipos.length-1];
	return str;
}
```

- **gmaps.js**

```
/**
 * Tiempo de recarga para el polling.
 */
var TIEMPO_POLLING = 5 * 1000;
/**
 * Opciones básicas del mapa.
 */
var myOptions;
/**
 * El mapa de GMaps.
 */
var map;
/**
 * Los límites del mapa.
 */
```

```
var bounds;
/**
 * Array de marcadores del mapa.
 */
var markersArray;
/**
 * Identificador del intervalo del polling.
 */
var intervalID;
/**
 * Criterios de búsqueda aplicados actualmente.
 */
var criterios;
/**
 * Iconos de tipo de incidencia activos o inactivos (para búsqueda).
 */
var iconos;

/**
 * Información estática de las incidencias, en función de su indice.
 * Recoge lo siguiente : [nombre,scaleX,scaleY,active,value]
 */
var incidencias;

/**
 * Inicializamos los parámetros del mapa.
 */
function initialize() {

	myOptions = {
		zoom : 15,
		mapTypeId : google.maps.MapTypeId.ROADMAP,
```

```
        };


        map = new
google.maps.Map(document.getElementById("map_canvas"),

                        myOptions);

        bounds = new google.maps.LatLngBounds();

        markersArray = [];


        var initialLocation;

        var almeria = new google.maps.LatLng(36.8418336,

                        -2.4550168000000667);


        navigator.geolocation.getCurrentPosition(function(position) {

                initialLocation = new
google.maps.LatLng(position.coords.latitude,

                                position.coords.longitude);

                map.setCenter(initialLocation);

        }, function() {

                alert("No se puede obtener la localizacion");

                initialLocation = almeria;

                map.setCenter(initialLocation);

        });


        incidencias = [

            ['Accidente',25,25,true,0],

            ['Veh\u00edculo abandonado',25,25,true,1],

            ['Congesti\u00f3n de tr\u00e1fico',25,25,true,2],

            ['Veh\u00edculo mal estacionado',25,25,true,3],

            ['Estacionamiento libre',25,25,true,4],

            ['Punto negro',25,25,false,5],

            ['Radar',25,25,false,6],

            ['Obras',25,25,true,7],

            ['Incidencia meteorol\u00f3gica',25,25,true,8],
```

```
		['Calle cortada',25,25,true,9],
		['Eventos',25,25,true,10],
		['Otros',25,25,true,11]
	];

	iconos = {};
	for (var i=0;i<incidencias.length;i++) {
		if (incidencias[i][3]) {
			iconos[i] = true;
		}
	}

	var inputUsuario = document.getElementById('usuario');
	inputUsuario.value = '';
	var inputTiempo = document.getElementById('tiempo');
	inputTiempo.value = 1;

}

/*
 * FUNCIONES BASICAS
 */

/**
 * Centramos y cambiamos el nivel de zoom para que se ajuste
 * a los marcadores actuales del mapa.
 */
function centrarMapa() {
	if (markersArray.length > 0)
		map.fitBounds(bounds);
}
```

```
/**
 * Realizamos la búsqueda avanzada según los parámetros seleccionados.
 * También fija los criterios de búsqueda para el polling.
 */
function buscar() {
	clearTimeout(intervalID);

	var tiposSeleccionados = [];
	var todos = true;
	var ninguno = true;
	for (var indice in iconos) {
		if (iconos[indice]) {
			tiposSeleccionados.push(indice);
			ninguno = false;
		}
		else {
			todos = false;
		}
	}
	if (ninguno) {
		clearMap();
		criterios = null;
	}
	else {
		if (todos)
			tiposSeleccionados = null;
		var inputUsuario = document.getElementById('usuario');
		var usuario = inputUsuario.value;

		var inputTiempo = document.getElementById('tiempo');
		var tiempo = inputTiempo.value;
```

```
			clearMap();

			criterios = {};
			criterios['tipos'] = tiposSeleccionados;
			criterios['usuario'] = usuario;
			criterios['tiempo'] = tiempo;

			loadSearch(tiposSeleccionados,usuario,tiempo);
	}
}

/*
 * FUNCIONES AUXILIARES
 */



/**
 * Cambiamos el estado de un icono de búsqueda (activo/inactivo)
 * @param indice El indice del icono a cambiar.
 */
function swapIcon(indice) {
	iconos[indice] = !iconos[indice];
	cambiarImagen(indice);
}

/**
 * Realizamos un cambio de imagen a un icono (activado/desactivado)
 * @param indice El indice del icono
 */
function cambiarImagen(indice) {
	var elem = document.getElementById("img_icon"+indice);
```

```
	var nombre = "iconos/icono"+indice;
	if (!iconos[indice]) {
		nombre += "desactivado";
	}
	nombre += ".png";
	elem.src = nombre;
}

/**
 * Activamos todos los iconos.
 */
function activarIconos() {
	for ( var indice in iconos) {
		iconos[indice] = true;
		cambiarImagen(indice);
	}
}

/**
 * Desactivamos todos los iconos.
 */
function desactivarIconos() {
	for ( var indice in iconos) {
		iconos[indice] = false;
		cambiarImagen(indice);
	}
}



/**
 * Limpiamos el mapa de marcadores.
 */
```

```
function clearMap() {
        for ( var i = 0; i < markersArray.length; i++) {
                markersArray[i].setMap(null);
        }
        bounds = new google.maps.LatLngBounds();
        markersArray = [];
}

/**
 * Añadimos un nuevo marcador al mapa.
 * @param marker El marcador a añadir.
 */
function addMarker(marker) {
        marker.setMap(map);
        markersArray.push(marker);
        bounds.extend(marker.getPosition());
}

/**
 * Función que se encarga de preparar el polling.
 */
function polling() {
        if (criterios != null)
                intervalID = setTimeout(prepareLoadNew,TIEMPO_POLLING);
}

/**
 * Preparamos los parámetros de búsqueda según los criterios
 * y llamamos al polling con AJAX.
 */
function prepareLoadNew() {
        var tipos = criterios['tipos'];
```

```
        var usuario = criterios['usuario'];
        var tiempo = criterios['tiempo'];

        loadNew(tipos,usuario,tiempo);
}
```

- **index.css**

```
html {
        height: 100%;
}

body {
        height: 100%;
        margin: 0px;
        padding: 0px;
}

ul li {
        display: inline-block;
        vertical-align: middle;
        padding: 3px;
        margin: 3px;
}

ul {
        text-align: center;
        position: relative;
        width: 100%;
        height: 100%;
        margin: 0px;
        padding: 0px;
        font-size: 15px;
}
```

```
.primero {
	margin-left : 1px;
	margin-right : 1px;
	padding-right : 1px;
	border-left: 1px solid orange;
}

.chico {
	margin-left : 1px;
	padding-left : 1px;
}



#search_bar {
	width: 100%;
	display: inline-block;
	height: 5%;
}

#map_canvas {
	width: 100%;
	height: 95%;
}

.boton {
	-moz-box-shadow: inset 0px 1px 0px 0px #fce2c1;
	-webkit-box-shadow: inset 0px 1px 0px 0px #fce2c1;
	box-shadow: inset 0px 1px 0px 0px #fce2c1;
	background: -webkit-gradient(linear, left top, left bottom, color-stop(0.05, #ffc477
			), color-stop(1, #fb9e25));
```

```
	background: -moz-linear-gradient(center top, #ffc477 5%, #fb9e25 100%);
	filter:
progid:DXImageTransform.Microsoft.gradient(startColorstr='#ffc477',
		endColorstr='#fb9e25');
	background-color: #ffc477;
	-webkit-border-top-left-radius: 20px;
	-moz-border-radius-topleft: 20px;
	border-top-left-radius: 20px;
	-webkit-border-top-right-radius: 20px;
	-moz-border-radius-topright: 20px;
	border-top-right-radius: 20px;
	-webkit-border-bottom-right-radius: 20px;
	-moz-border-radius-bottomright: 20px;
	border-bottom-right-radius: 20px;
	-webkit-border-bottom-left-radius: 20px;
	-moz-border-radius-bottomleft: 20px;
	border-bottom-left-radius: 20px;
	text-indent: 0;
	border: 1px solid #eeb44f;
	display: inline-block;
	color: #ffffff;
	font-family: Arial Black;
	font-size: 15px;
	font-weight: bold;
	font-style: normal;
	height: 20px;
	line-height: 20px;
	width: 100px;
	text-decoration: none;
	text-align: center;
	text-shadow: 1px 1px 0px #cc9f52;
}
```

```
.boton:hover {
	background: -webkit-gradient(linear, left top, left bottom, color-stop(0.05, #fb9e25
		), color-stop(1, #ffc477));
	background: -moz-linear-gradient(center top, #fb9e25 5%, #ffc477 100%);
	filter: progid:DXImageTransform.Microsoft.gradient(startColorstr='#fb9e25',
		endColorstr='#ffc477');
	background-color: #fb9e25;
}

.boton:active {
	position: relative;
	top: 1px;
}

.desactivado {
	pointer-events:  none;
	cursor: default;
	background:-webkit-gradient( linear, left top, left bottom, color-stop(0.05, #fae4bd), color-stop(1, #eac380) );
	background:-moz-linear-gradient( center top, #fae4bd 5%, #eac380 100% );
	filter:progid:DXImageTransform.Microsoft.gradient(startColorstr='#fae4bd', endColorstr='#eac380');
}
```