DGTE-006: Desarrollo de aplicaciones para dispositivos móviles

Versión 1.1

Área de Arquitectura

GERENCIA INFORMÁTICA
JOSEFA VALCÁRCEL, 44
28027-MADRID

Contenido

Índice de Ilustraciones

1 Introducción

1.1 Objetivo

Durante los últimos años se ha producido un auge en el uso de dispositivos móviles, que ha llevado a la evolución y desarrollo tanto de los propios dispositivos como de las plataformas sobre las cuales se ejecutan aplicaciones en los mismos. Dichas aplicaciones han permitido tanto la extensión del uso de Internet en dispositivos móviles como el aprovechamiento de las posibilidades que tales dispositivos ofrecen de forma nativa (GPS, sensores, cámaras de foto y video…) para ofrecer funcionalidades encaminadas tanto al ocio y la domótica como al trabajo, constituyéndose los dispositivos móviles en herramientas de uso cotidiano que permiten realizar fuera del hogar y de la oficina funciones que anteriormente exigían la presencia física en un lugar determinado. En este contexto se hace necesario ofrecer, tanto al personal de la Dirección General de Tráfico como a los ciudadanos en general y a organismos y/o empresas colaboradoras, aplicaciones para dispositivos móviles que contribuyan a aumentar la productividad y a facilitar y extender el uso de la administración electrónica. En ese sentido, es necesario asegurar la coherencia y homogeneidad en todo desarrollo para dispositivos móviles. Con esta especificación la Dirección General de Tráfico pretende proporcionar el conjunto de tecnologías y patrones que deberán utilizarse durante la construcción de aplicaciones para dispositivos móviles.

Los objetivos de los patrones contenidos en este documento son:

- Establecer la reutilización de implementaciones (lógica de negocio, conectores, servicios técnicos…) entre diferentes aplicaciones móviles

para disminuir la necesidad de volver a implementar esos artefactos, rebajando la posibilidad de introducir errores en el proyecto.

- Establecer las directivas de arquitectura para las aplicaciones destinadas a ejecutarse en dispositivos móviles de manera que se minimice el acoplamiento y se maximice la escalabilidad de los desarrollos realizados.

- Establecer los mecanismos adecuados de integración con los sistemas *backend* necesarios.

- Establecer los mecanismos de seguridad que protejan tanto el intercambio de datos entre los dispositivos móviles y los sistemas *backend* como la información sensible que tenga que custodiarse en los dispositivos móviles durante el mínimo tiempo que sea necesario.

- Establecer los patrones de diseño más convenientes para garantizar un equilibrio adecuado entre rendimiento y flexibilidad, en aras de disminuir los costes de mantenimiento.

1.2 Destinatarios del documento

Este documento está por tanto dirigido a todos aquellos arquitectos, analistas y desarrolladores de aplicaciones para dispositivos móviles para la Dirección General de Tráfico.

El documento presupone que el lector tiene familiaridad con los diferentes lenguajes de programación para las diferentes plataformas móviles homologadas por la Dirección General de Tráfico.

Ámbito tecnológico	Plataforma	Lenguaje
Nativo	Android	Java, "Kotlin/JVM"
Nativo	iOS	Objective-C, Swift
Cross Platform	Phonegap	Javascript, CSS3, HTML5

1.3 Convenciones tipográficas

Se mostrarán en *cursiva* aquellas expresiones técnicas en idioma inglés que no se han traducido bien porque no existe un término adecuado en español, bien porque son expresiones sobradamente conocidas en su idioma nativo por los destinatarios del documento y por lo tanto resulta más conveniente no traducirlas en aras de mejorar la comprensión del texto.

La fuente Helvética se usa para el grueso del texto, mientras que la fuente Palatino se usa para títulos y contenido de tablas. La fuente `Courier New` se usa para los ejemplos de código, los cuales se ofrecen en cuadros sombreados.

1.4 Convenios de la especificación

En la definición de los nombres de paquetes se ha usado una terminología para referirse a ciertas partes de la estructura que dependen del proyecto en concreto y del ámbito funcional de las clases que formarán parte de los mismos.

Area_De_Negocio	Se refiere al área de negocio a la que, por funcionalidad, correspondería la aplicación (vehículos, conductores, sanciones, sic, arquitectura, exámenes, observatorio).
Acronimo_Proyecto	Se refiere al acrónimo de cuatro letras que se establecerá para cada aplicación desarrollada.

Nombre_Subsistema	Si existe más de un subsistema, se refiere al nombre del subsistema en que se ubica el paquete.
Tópico	Significa un grupo funcional o agrupación de operaciones relacionadas dentro de una temática común.
Nombre_Paquete	Se refiere al nombre de paquete que cada equipo de desarrollo elegirá.

Asimismo, se han establecido unos convenios para indicar qué partes son opcionales y cuáles pueden repetirse.

[]	Representa un elemento opcional.
*	Indica que un elemento puede repetirse

2 Diseño de la Arquitectura

El diseño de Arquitectura, independientemente del ámbito tecnológico considerado en el proyecto (Nativo, Híbrido o Web móvil), se establecerá a través de un modelo de n-capas. Dichas capas estarán claramente separadas y tendrán que ser fáciles de mantener, así como ser intercambiables por otra implementación de la capa sin afectar al código escrito en las otras capas. Para permitir esto último se trabajará con interfaces para la comunicación entre capas. El modelo de capas se basará en el patrón MVC (Modelo-Vista-Controlador) y en el patrón de arquitectura hexagonal (también llamado de puertos y adaptadores).

Se definirán las siguientes capas: Presentación, Negocio, Servicios (adaptadores de integración y servicios técnicos) y Persistencia.

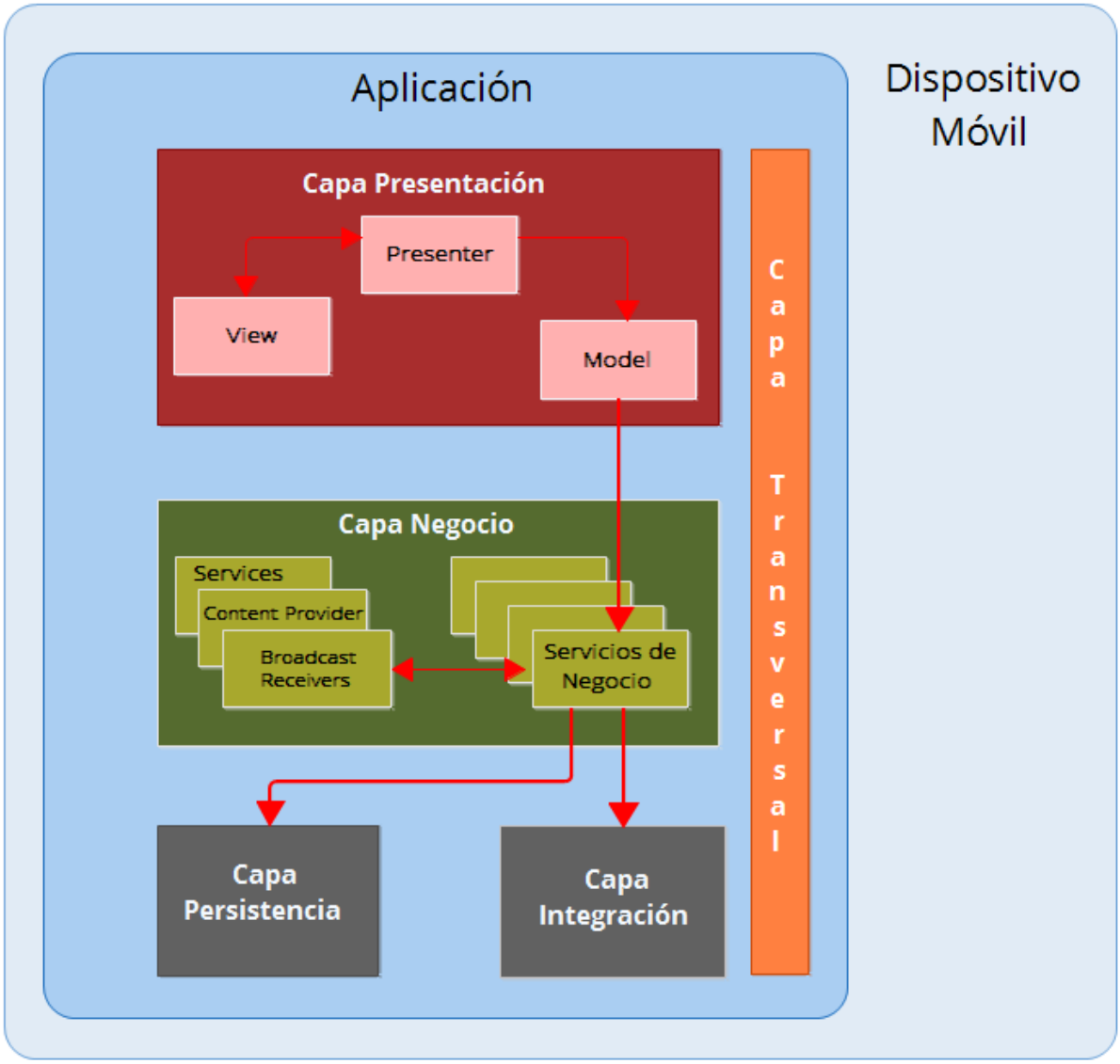


Ilustración 1: Arquitectura de capas desarrollo móvil

2.1 Capa de Presentación

La implementación de la capa de presentación dependerá del ámbito tecnológico planteado en el proyecto (Nativo, Híbrido, Web móvil). Como regla general se usarán, para la modelización de la capa de presentación, todos aquellos artefactos

definidos en el Framework o SDK asociado a la tecnología objetivo usada en el proyecto.

Los componentes que conforman esta capa, de manera genérica e independiente del ámbito tecnológico usando, pueden separarse en dos grupos:

- **Vistas o elementos visuales:** Son los elementos más básicos que conforman una vista y que permiten la interacción por parte del usuario. Son auto contenidos en lo que a la gestión de eventos se refiere y contienen las propiedades básicas para su particularización visual. Dentro de esta agrupación se deberá diferenciar los elementos de control y los elementos de navegación. Los elementos de control son aquellos elementos interactivos de la interfaz de usuario que permiten lanzar eventos a nivel de capa de aplicación y que por lo tanto están en mayor o menor grado interrelacionados con la lógica de negocio de la aplicación. En este grupo se encontrarían, por ejemplo: B*utton*, C*heckbox*, *Text Field*, R*adio Button, Toggle Button, Spinner, Pickers…* Por otra parte, los elementos de navegación son aquellos cuya única responsabilidad es la gestión de la navegación dentro de las vistas de la aplicación. En este grupo se encontrarían, por ejemplo: *NavigationBar*, *Segmented Control*, *TabBar,* etc.

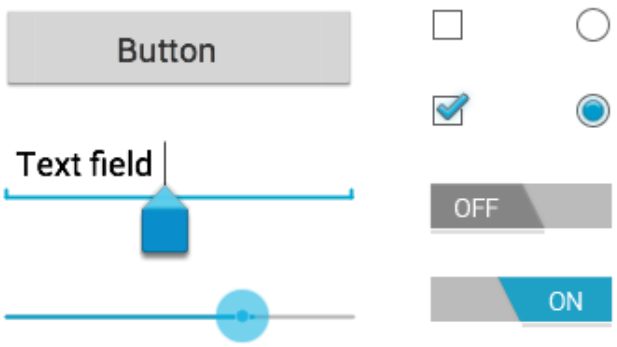


Ilustración 2: Ejemplo de elementos de control

Los elementos visuales disponibles para las diferentes tecnologías móviles (Android, iOS, Windows Phone…) pueden ser consultados en sus respectivas documentaciones técnicas.

Tecnología	Consulta de elementos visuales disponibles
Android	http://developer.android.com/guide/topics/ui/controls.html
iPhone	https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/Bars.html#//apple_ref/doc/uid/TP40006556-CH12-SW1
Windows Phone	https://msdn.microsoft.com/library/windows/apps/xaml/hh465351.aspx

Se deberá tener en cuenta que todos aquellos elementos visuales desarrollados programáticamente también se englobarán dentro de esta categoría y deberán seguir todas las recomendaciones aplicables.

- **Agrupadores de Vistas o Contenedores:** Son los elementos responsables de la estructuración en las vistas mediante la agrupación de elementos visuales u otros agrupadores en forma de árbol lógico.

Normalmente este tipo de elementos no tienen representación gráfica y centran sus opciones de configuración a aquellas relativas al posicionamiento relativo de sus elementos (o de ellos mismos) respecto a otros agrupadores / elementos visuales presentes en la vista.

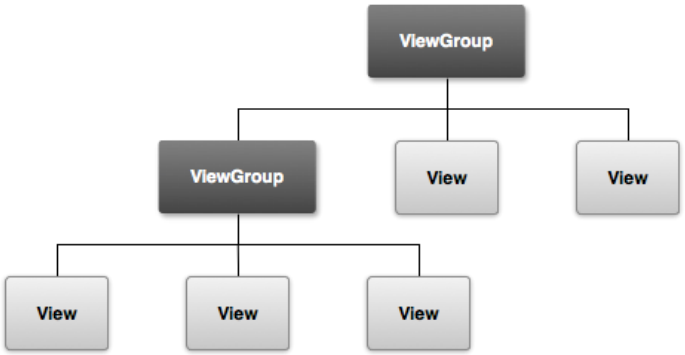


Ilustración 3: Estructura de árbol lógico de los diferentes componentes en una vista

Para el desarrollo de la capa de presentación se deberá tener en cuenta las siguientes directrices:

Ámbito tecnológico	Tecnología móvil	Observaciones
Nativo	Android	Las definiciones de las vistas se basarán en el uso de XML, tal como se especifica en la documentación de la plataforma: *http://developer.android.com/guide/topics/ui/declaring-layout.html*
Híbrido	Phonegap	Las definiciones de las vistas se basarán en el uso de HTML5 y CSS3, tal como se especifica en: *http://phonegap.com/2012/05/02/phonegap-explained-visually/*

Web móvil	HTML5 y CSS3	Las definiciones de las vistas se basarán en el uso de HTML5 y CSS3, así como en la librería de componentes PrimeFaces Mobile (PMF). PrimeFaces Mobile (PFM) es un kit de interfaz de usuario para crear aplicaciones JSF optimizadas para dispositivos móviles. PFM se construye sobre jQuery Mobile, un marco HTML5 de interfaz de usuario que proporciona soporte táctil optimizado para varias plataformas. Además de la perfecta integración con jQuery Mobile, PFM cuenta con un RenderKit móvil para los componentes populares Primefaces (en su versión web), extensiones framework Ajax, eventos de comportamiento ajax móviles, modelo de navegación integrado, de carga lenta de las páginas, widgets...

2.1.1 Organización de la Capa de Presentación en Android: Patrón MVP.

El MVP (Model View Presenter) es un patrón derivado del MVC (Model View Controller) que permite separar la capa de presentación de la lógica de la misma, separando el funcionamiento de la interfaz de su forma de representarlo en pantalla, pudiendo tener diferentes vistas con una misma lógica.

Debido a que los componentes principales de Android (Activities y Fragments) para implementar la capa de presentación, están acoplados tanto con la interfaz como con la lógica de obtención de los datos, el uso del patrón MVP nos facilitará separar la vista de la lógica de la vista. Dicha separación nos facilitará la realización de test unitarios y la posibilidad de realizar tareas en segundo plano que al no estar conectadas a una Activity serán independientes del ciclo de vida de ésta y no consumirá memoria del hilo principal (UI Thread).

Aplicando este patrón, la capa de presentación se dividirá a su vez en las siguientes subcapas:

- **Presentador:** Responsable de:
    - Actuar de intermediario entre la vista y el modelo.
    - Decidir la acción que se desencadenará cuando se interactúa con la vista.
    - Enviar y recibir eventos a través de un patrón Event Bus.
- **Vista:** Responsable de mostrar la información e invocar a un método del Presenter cuando se produzca un evento sobre la interfaz. La vista, será implementada por una Activity o Fragment que contendrá una referencia al presentador. En ningún caso se deberá crear vistas en tiempo de ejecución desde cualquier otra capa.
- **Modelo:** Responsable de obtener los datos que se deben mostrar en la vista. El acceso a los Servicios de Negocio se deberá realizar siempre mediante el uso Factory o de los patrones Inversion of Control (IoC), y siempre a través de las instancias definidas para los Servicios de Negocio. En ningún caso se deberá instanciar Objetos correspondientes a los Servicios de Negocio. Los únicos accesos permitidos en la implementación de esta capa será a los Servicios de Negocio definidos en la capa de negocio. En el caso excepcional en que se deba acceder a un artefacto definido en una capa diferente a la capa de negocio deberá solicitarse la justificación al Departamento de Arquitectura previo a la entrega del diseño.

2.1.2 Gestion de Eventos:

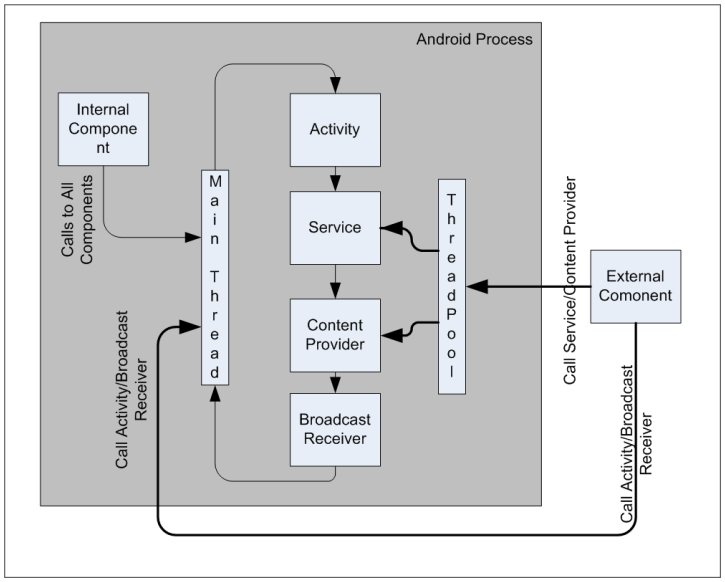


Ilustración 4: Diagrama del ciclo de vida de un proceso Android

La gestión de los eventos (tanto internos – Intents –, como externos – Broadcast -) es responsabilidad del *Looper*, del cual existe una única instancia por instancia de la máquina virtual dalvik (DVM), y se ejecuta en el hilo principal de la aplicación *UI Thread*. Es por ello por lo que hay que tener un especial cuidado en la implementación de las funciones de *callback* asociados a los eventos de manera que no se bloquee el hilo principal. Para ello, la recomendación es propagar dichos eventos a un bus de eventos. Se permitirá utilizar la implementación de un bus de eventos mediante la librería externa ***"Square Otto"***.

Nombre	Descripción
Square Otto	http://square.github.io/otto/

La estructura de paquetes que cubren la capa de Presentación son:

Paquete Descripción
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].pres.view.component.[Nombre_Paquete]* Paquete que contiene los componentes visuales desarrollados específicamente para el proyecto.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].pres.[Topico].view.[Nombre_Paquete]* Paquete que contendrá las interfaces de vista
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].pres.[Topico].view.[Nombre_Paquete]*.impl Paquete que contendrá las implementaciones de la vista (Activities, Fragments)
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].pres.[Topico].view.[Nombre_Paquete]*.adapter Paquete que contendrá los adaptadores
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].pres.[Topico].presenter.[Nombre_Paquete]* Paquete que contendrá las interfaces de los presentadores
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].pres.[Topico].presenter.[Nombre_Paquete]*.impl Paquete que contendrá las implementaciones de los presentadores
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].pres.util.[Nombre_Paquete]* Paquete que contendrá todos los elementos de la capa de presentación que por su naturaleza no se adaptan a ninguno del resto de paquetes definidos para esta capa.

2.2 Capa de Negocio

Se opta por una solución basada en servicios que encapsulen la implementación de la lógica de negocio de la aplicación. Cada servicio definido en la capa de negocio se deberá componer de una interfaz y de una clase que implemente los métodos definidos en la interfaz.

La instanciación o referencia al servicio se realizará mediante el uso del Patrón de *Inversion Of Control (IoC)* o mediante el uso del Patrón *Factory*.

En el caso de usar *Inversion Of Control (IoC)* se permitirá el uso de las siguientes librerías:

Nombre	Descripción
Butterknife	http://jakewharton.github.io/butterknife/
Dagger	http://github.com/square/dagger

El acceso al resto de capas (persistencia y/o integración) se deberá realizar a través de los mismos patrones mencionados anteriormente (*IoC* o *Factory*) y siempre a través de sus respectivas interfaces. Queda por tanto completamente prohibida la instanciación directa de objetos del resto de capas en las implementaciones de los servicios de negocio. El objetivo es que exista el máximo desacoplamiento entre la implementación de la lógica de negocio y la implementación de aquellos artefactos que proporcionen funcionalidades de integración o persistencia.

A la hora de diseñar los servicios de negocio se deberá tener en cuenta el ámbito de responsabilidad de los mismos. En este caso se debería considerar una estrategia que maximice la atomicidad de las responsabilidades, de manera que se consiga una alta reusabilidad y cohesión del código con un mínimo acoplamiento entre los diferentes servicios de negocio.

Para conseguir un rendimiento óptimo y evitar problemas derivados del bloqueo del *UI Thread* los servicios de negocio se deberán diseñar de manera que observen las siguientes consideraciones:

- Si el servicio de negocio implementa lógica de negocio que requiere operaciones intensivas de CPU (cálculos complejos, gráficos en 3D…) se deberán diseñar para que su implementación se ejecute en un *Worker Thread* a través del uso de las estrategias establecidas para cada una de las plataformas.

- Si el servicio de negocio implementa lógica de negocio que requiere de artefactos de integración, persistencia u otros que contengan operaciones de I/O sobre el sistema, se deberá observar si la implementación de dichos artefactos se realiza sobre un *Worker Thread*. En caso contrario, la implementación de la lógica de negocio deberá realizarse sobre un *Worker Thread* para mantener una estrategia de protección sobre bloqueos del *thread* de la vista.

En la capa de Negocio también se ubicarán las implementaciones de los Servicios *(Services)*, Proveedores de Contenido *(Content Provider)* y los *Broadcast Receivers.*

La implementación de los mismos deberá seguir las recomendaciones proporcionadas en:

Ámbito tecnológico	Tecnología móvil	Tecnología servicio	Recomendaciones
Nativo	Android	Content Provider	*http://developer.android.com/guide/topics/providers/content-providers.html*
Nativo	Android	Broadcast Receiver	*http://developer.android.com/reference/android/content/BroadcastReceiver.html*
Nativo	Android	Services	*http://developer.android.com/guid*

e/components/services.html

La estructura de paquetes que cubren la capa de Negocio son:

Paquete Descripción
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].neg.[Nombre_Paquete]* Paquete que contiene todas las interfaces de negocio.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].neg.impl.[Nombre_Paquete]* Paquete que contiene todas aquellas clases que implementen las interfaces de negocio.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].neg.receiver.[Nombre_Paquete]* Paquete que contiene todas las implementaciones de receptores de eventos (Broadcast Receiver) de la aplicación.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].neg.service.[Nombre_Paquete]* Paquete que contiene todos los servicios (Services) definidos en el proyecto.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].neg.contentprovider.[Nombre_Paquete]* Paquete que contiene todas las implementaciones de los proveedores de contenido (*Content Provider*) proporcionados por la aplicación.

2.3 Capa de Adaptadores de Servicio (Integración)

Se deberá distinguir entre los cliente de integración correspondientes a servicios publicados externamente al dispositivo móvil (integración externa) y aquellos servicios publicados internamente en el dispositivo móvil (integración interna) y proporcionados por otras aplicaciones instaladas en el mismo dispositivo.

En el primer caso (integración externa) la definición e implementación de los clientes dependerá tanto del ámbito tecnológico del proyecto móvil (nativo, híbrido, web móvil) así como de la tecnología en la que se encuentran implementados dichos servicios (SOAP basados en WS-I o REST/JSON).

Por ello se deberán seguir las directrices contenidas en la siguiente tabla:

Ámbito tecnológico	Tecnología móvil	Tecnología servicio	Librería homologada
Nativo	Android	SOAP WS-I	Librería kSOAP2 o Servicio http://easywsdl.com
Nativo	Android	REST/JSON	Librería Retrofit Librería Volley
Nativo	iOS	SOAP WS-I	Librería SOAPEngine o Servicio http://easywsdl.com/
Nativo	iOS	REST/JSON	Librería RestKit
Híbrido	Phonegap	SOAP WS-I	Librería JQuery.Soap
Híbrido	Phonegap	REST/JSON	Librería JQuery Mobile ($.ajax)
Web móvil	HTML5 / CSS / PrimeFaces	SOAP WS-I	Librería JQuery.Soap
Web móvil	HTML5 / CSS / PrimeFaces	REST/JSON	Librería JQuery Mobile ($.ajax)

La integración se realizará, siempre que sea posible, a través de la tecnología REST/JSON. La integración con servicios SOAP WS-I siempre se deberá realizar previa autorización del Departamento de Arquitectura.

Existe un ESB que intermediará las peticiones a los servicios externos ofreciendo una interfaz REST con independencia de la tecnología con que el servicio externo

esté disponible en su origen. Toda invocación se realizará aportando las credenciales necesarias para el acceso al servicio externo.

Se usarán las librerías homologadas (Retrofit o Volley para REST/JSON y kSOAP2 para SOAP WS-I). La documentación para la creación de los clientes puede accederse en:

Librería homologada	Descripción
Retrofit	http://square.github.io/retrofit/
Volley	http://developer.android.com/training/volley/index.html
kSOAP2	https://code.google.com/p/ksoap2-android/

Para cubrir escenarios de NO conectividad se permitirá el uso de la librería “Path Job Queue” para gestionar tareas en background por prioridad y/o posponer su ejecución hasta que se recupere la conectividad

Librería homologada	Descripción
Path Job Queue	https://github.com/path/android-priority-jobqueue

En el supuesto de que se requiera realizar una implementación directamente a través del SDK de Android, se seguirán las siguientes indicaciones:

- La implementación de la comunicación (acceso al servicio) se deberá implementar directamente con las clases disponibles en el SDK de Android. No se podrán usar, por tanto, implementaciones de terceros (por ejemplo *Apache HTTPComponents*).

- Toda la implementación de la lógica del cliente del servicio se deberá desarrollar en un *AsyncTask* o en un *Thread*. No se podrá realizar una implementación directamente en una *Activity*, *Fragment*, *Service*, objeto de la capa de negocio o cualquier otro componente que se ejecute en el *UI Thread.*

- Toda la implementación de la lógica del cliente del servicio deberá pensarse para ejecutarse de manera asíncrona y mediante el paradigma de *event-driven development* (la implementación del manejo de eventos se realizará utilizando la librería "Square Otto"). Será por tanto obligatorio definir como mínimo una interfaz por servicio, que será la responsable de proveer el desacoplamiento necesario tanto en las peticiones a los servicios definidos, así como en la gestión de la respuesta de los mismos (a través de la definición de funciones de *callback*).

En el segundo caso (integración interna) el diseño e implementación de los clientes dependerá exclusivamente del ámbito tecnológico del proyecto móvil (nativo, híbrido, web móvil).

Por ello se seguirán las directrices contenidas en la siguiente tabla:

Ámbito tecnológico	Tecnología móvil	Tecnología servicio	Recomendaciones
Nativo	Android	Content Provider	*http://developer.android.com/guide/topics/providers/content-providers.html*
Nativo	Android	Broadcast Intent	*http://developer.android.com/guide/components/intents-filters.html*

Nativo	Android	Intent & Intent Filters	*http://developer.android.com/guide/components/intents-filters.html*
Nativo	Android	Custom URL schemas	*https://developer.android.com/guide/components/intents-common.html*

Independientemente del tipo de integración que se realice (interna o externa) se deberá tener en cuenta, tanto en su definición como en su implementación, que dicha integración en ningún caso debería bloquear el hilo de ejecución de la aplicación (también conocido como *UI Thread o Web Thread*). Asimismo se construirán adaptadores para las invocaciones a servicios de forma que el negocio quede desacoplado de los artefactos tecnológicos concretos que sea necesario utilizar para realizar la invocación a dichos servicios. Esto es extensible a las utilidades que la aplicación necesite utilizar (librerías, *frameworks*) de forma que la aplicación no quede acoplada a los servicios de infraestructura o a implementaciones concretas.

En la implementación de servicios en *backend* que den servicio a aplicaciones móviles se deberá tener en cuenta las siguientes observaciones:

- Estructuras de datos optimizados para el procesado en dispositivos con recursos limitados. El motivo de esta necesidad es que, a pesar de los avances que se van produciendo, los dispositivos móviles todavía se consideran por definición dispositivos con recursos (CPU, memoria, batería…) limitados y finitos (conectividad a través tarifa de datos). Es por ello que la optimización de todos los procesos y en concreto aquellos relacionados con el procesado de la información (integración, persistencia…) debe ser un punto importante a tener en cuenta. Es por

ello que un punto inicial a tener en cuenta en la optimización que se ha comentado es la estructura de datos utilizada. De esta manera el impacto de menor a mayor grado corresponde a JSON, XML, SOAP respectivamente. Se deberá, por lo tanto, usar siempre aquellas estructuras de datos que tengan un menor impacto en el procesamiento, con prioridad al uso de JSON considerando el uso de SOAP y/o XML en casos justificados.

La estructura de paquetes que cubren esta capa es la siguiente:

Paquete Descripción
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].adaptador.[Nombre_Paquete]*.impl.[Nombre_Paquete]* Paquete que contiene las clases de implementación de las llamadas a los servicios internos y clientes de los servicios externos que se integrarán en la aplicación.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].adaptador.[Nombre_Paquete]* Paquete que contiene todas las interfaces asociadas a los servicios internos y/o externos (REST/JSON) que se integrarán en la aplicación.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].adaptador.[Nombre_Paquete]*.endpoints Paquete que contiene todas las clases de constantes que definan los endPoints de los servicios externos con los que se integrarán en la aplicación

2.4 Capa de Persistencia

Se usará una solución de ORM en la implementación de la capa de persistencia. A continuación se detallan las diferentes soluciones por tipo de plataforma:

Nombre	Plataforma	Descripción

GreenDAO	Android	http://greendao-orm.com/

Estas soluciones permiten realizar el mapeo objeto-relacional sin tener que implementarlo directamente en JDBC, a la vez que nos desacopla de la base de datos subyacente. Es interesante poder seguir teniendo acceso a la capa inmediatamente inferior para poder tener acceso a la funcionalidad que no aporte la solución, además de poder usar directamente SQL si se considera necesario por cuestiones de funcionalidad o rendimiento.

La estructura de paquetes que cubren esta capa es la siguiente:

Paquete Descripción
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema**].dao.[Nombre_Paquete]*** Paquete que contiene las interfaces de acceso a datos.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema**].dao.impl.[Nombre_Paquete]*** Paquete que contiene las implementaciones basadas en el ORM de las interfaces de acceso a datos.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema**].dao.entidades.[Nombre_Paquete]*** Paquete que contiene todas los *beans* correspondiente al modelo de datos usados en las transacciones con las implementaciones DAO.

2.5 Capas Transversales

En el modelo de Arquitectura propuesto en este documento para el desarrollo de proyectos para dispositivos móviles no se contempla la obligatoriedad de tener en cuenta en el diseño y construcción de capas transversales (por ejemplo: seguridad, transaccionalidad…) con una implementación accesible desde las demás definidas en el modelo (controladores, negocio, integración).

Este tipo de capas se podrán (o deberán) añadir según las necesidades de cada aplicación y deberá quedar convenientemente justificado en la documentación de la arquitectura de la aplicación.

La implementación de estas capas se basará en el uso de *beans* específicos y en ningún caso se implementarán componentes visuales, de control, navegación, persistencia o integración dentro de esta capa. En su lugar, se implementará en las capas correspondientes según su naturaleza y las descripciones proporcionadas en cada uno de los puntos anteriores.

La estructura de paquetes que cubren esta capa es la siguiente:

Paquete Descripción
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].util.[Nombre_Paquete]* Contendrá todas aquellas clases de utilidad clasificadas por funcionalidad.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].ioc.[Nombre_Paquete]* Contendrá todas aquellas necesarias para realizar la implementación de inyección de dependencias.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].modelo.[Nombre_Paquete]* Contendrá todas aquellas clases que componen el dominio interno de la aplicación.
es.trafico.Acronimo_Proyecto.[Nombre_Subsistema].dominio.[Nombre_Paquete]* Contendrá todas aquellas clases que componen el dominio DGT de la aplicación.

A continuación se ofrece un diagrama tipo para una aplicación que necesite todas las capas y descomponga funcionalmente su negocio interno.

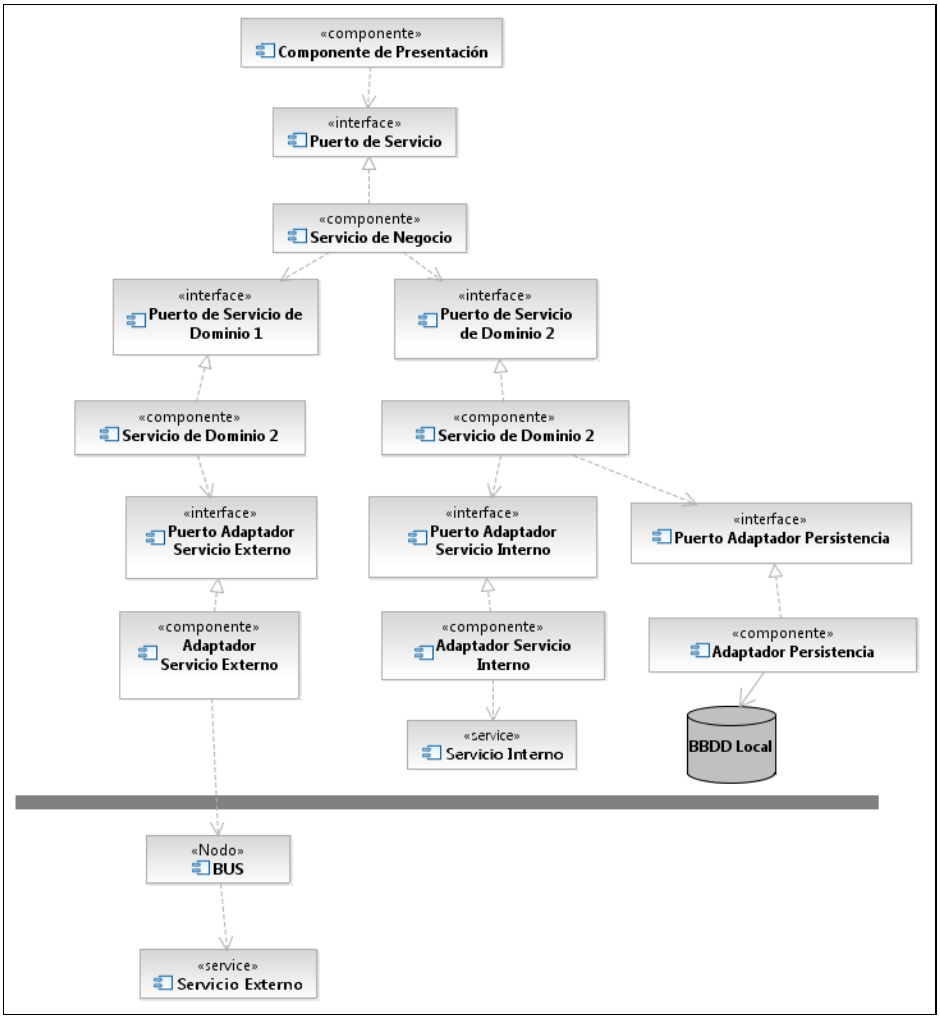


Ilustración 5. Diagrama de Arquitectura tipo

3 Patrones de diseño generales

A continuación se detallan las consideraciones que deben tenerse en consideración en el desarrollo de aplicaciones para dispositivos móviles.

Deberá tenerse en cuenta las comprobaciones que se realizarán en las revisiones de código del proyecto y que son las siguientes:

- Cumplimiento de las reglas checkstyle, PMD y copy/paste.
- Cumplimiento de los principios de diseño y arquitectura reflejados en el presente documento y en la Guía de Desarrollo. Estos se verificarán mediante inspecciones de código tanto automatizadas como no automatizadas.
- Revisión de la completitud de la documentación del código, que se revisa a través del Javadoc del proyecto. Se examinará, por ejemplo, que todos los paquetes, clases, interfaces, métodos, etc… estén documentados siguiendo las directivas recogidas en el documento de normas de codificación.
- Trazabilidad entre diseño y código.
- La aplicación se deberá poder compilar sin ningún tipo de error.
- Las dependencias indicadas en los diagramas de paquetes que se hayan reflejado en el diseño serán las que se encontrarán realizadas en el código. La resolución de dichas dependencias se realizará

siempre a través del repositorio de artefactos y no a través del repositorio local.

Para obtener información más detallada sobre las comprobaciones de una auditoría consultar el documento "*Plantilla Informe Auditoría*".

3.1 Patrones de diseño. *Singleton*

Las clases Singleton representan objetos para los cuales solo debe existir una única instancia. Su uso es recomendable, por ejemplo, en clases que mapeen recursos que no vayan a variar en el tiempo (como ficheros de configuración) y que requieran ser accedidos desde diferentes puntos de la implementación de negocio (servicios de negocio).

```
public class Cliente {
          public static Cliente _instance;
          // constructor
          public static Cliente Get_Instance() {
                    if (_instance == null) {
                              _instance = new Cliente();
                    }
                    return _instance;
          }
          private Cliente(){
          }
}
```

3.2 Patrones de diseño. *Abstract Factory*

El uso de referencias a interfaces, en lugar de referencias a instancias de clases concretas es una buena práctica para evitar el acoplamiento (efecto dominó) ya que, de esa manera, se protege la implementación antes posibles cambios.

El patrón *Abstract Factory* define la creación de una familia de objetos relacionados sin conocimiento específico de la implementación concreta de cada uno de ellos.

```
//Our AbstractProduct
public interface Window {
          public void setTitle(String text);
          public void repaint();
}

// ConcreteProductA1
final class MSWindow implements Window {
          @Override
          public void repaint() {
                    // MS Windows specific behaviour
          }
          @Override
          public void setTitle(String text) {
                    // TODO Auto-generated method stub
          }
}

// ConcreteProductA2
final class MacOSXWindow implements Window {
          @Override
          public void repaint() {
                    // Mac OSX specific behaviour
          }
          @Override
          public void setTitle(String text) {
                    // TODO Auto-generated method stub
          }
}

// Our abstract factory
interface AbstractWidgetFactory {
          public Window createWindow();
}

// ConcreteFactory1
final class MsWindowsWidgetFactory {
          // create an MSWindow
          public Window createWindow() {
                    MSWindow window = new MSWindow();
                    return window;
          }
}

// ConcreteFactory2
final class MacOSXWidgetFactory {
          // create a MacOSXWindow
          public Window createWindow() {
                    MacOSXWindow window = new MacOSXWindow();
                    return window;
          }
}

// Client
final class GUIBuilder {
          public void buildWindow(AbstractWidgetFactory widgetFactory) {
                    Window window = widgetFactory.createWindow();
                    window.setTitle("New Window");
          }
}

//using GUIBuilder and saying which king of Window create
final class MainClass {
          public static void main(String[] args) {
                    GUIBuilder builder = new GUIBuilder();
                    AbstractWidgetFactory widgetFactory = null;
                    // check what platform we're on
                    if (Platform.currentPlatform() == "MACOSX") {
                              widgetFactory = new MacOSXWidgetFactory();
                    } else {
                              widgetFactory = new MsWindowsWidgetFactory();
                    }
                    builder.buildWindow(widgetFactory);
```

```
        }
}
```

3.3 Patrones de diseño. *Factory Methods.*

Los Factory Methods son métodos estáticos que devuelven una instancia de clase y cumplen con las siguientes características:

- Tienen nombres diferentes a los constructores
- No necesitan instanciar un objeto nuevo en cada llamada
- Pueden devolver un subtipo al tipo que retornan. Esto es muy útil cuando se devuelven interfaces.
- Los nombre más comunes para este tipo de métodos son *getInstance()* y *valueOf()* sin que sea obligatorio el uso de ninguno de los dos.

```
public final class ComplexNumber {
        /**
         * Static factory method returns an object of this class.
         */
        public static ComplexNumber valueOf(float aReal, float aImaginary) {
                return new ComplexNumber(aReal, aImaginary);
        }
        /**
         * Caller cannot see this private constructor.
         * The only way to build a ComplexNumber is by calling the static factory
         * method.
         */
        private ComplexNumber(float aReal, float aImaginary) {
                fReal = aReal;
                fImaginary = aImaginary;
      }
        private float fReal;
        private float fImaginary;
}
```

3.4 Patrones de diseño. *ViewHolder.*

El patrón *ViewHolder* es usado en las implementaciones de los *Adapters* para conseguir minimizar las búsquedas de las referencias a los elementos gráficos y por lo tanto mejorar los tiempos de respuesta y la experiencia de usuario.

Para ello se almacenará las referencias a los diferentes elementos gráficos que conforman del *layout (*y que se hayan recuperado una vez) dentro del mismo *layout* a través del método *View.setTag(Object).*

```
public class ViewHolder {
          @SuppressWarnings("unchecked")
          public static <T extends View> T get(View view, int id) {
              SparseArray<View> viewHolder = (SparseArray<View>) view.getTag();
              if (viewHolder == null) {
                  viewHolder = new SparseArray<View>();
                  view.setTag(viewHolder);
              }
              View childView = viewHolder.get(id);
              if (childView == null) {
                  childView = view.findViewById(id);
                  viewHolder.put(id, childView);
              }
              return (T) childView;
          }
}


@Override
private View getView(int position, View convertView, ViewGroup parent) {
          if (convertView == null) {
              convertView = LayoutInflater.from(context)
                .inflate(R.layout.banana_phone, parent, false);
          }

          ImageView bananaView = ViewHolder.get(convertView, R.id.banana);
          TextView phoneView = ViewHolder.get(convertView, R.id.phone);

          BananaPhone bananaPhone = getItem(position);
          phoneView.setText(bananaPhone.getPhone());
          bananaView.setImageResource(bananaPhone.getBanana());

          return convertView;
}
```

3.5 Patrones de diseño. *Observer.*

El patrón *Observer* define una dependencia de uno a muchos entre objetos de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y se actualizan automáticamente.

Los objetos que están viendo los cambios de estado se llaman observador. Alternativamente a observador también se le suele conocer como oyente. El objeto que está siendo observado se denomina sujeto.

Ejemplo: *View A* es el sujeto. *View A* muestra la temperatura de un recipiente. *View B* muestra una luz verde es la temperatura está por encima de 20 grados centígrados. Por lo tanto *View B* se registra a sí mismo como un oyente para *View A*. Si se cambia la temperatura de *View A* se activa un evento. Eso es un evento que se envía a todos los detectores registrados en este ejemplo *(View B)*. *View B* recibe los datos modificados y puede ajustar su contenido.

```
public class ObjectObserverPattern extends Activity implements Observer, OnClickListener {
        BaseApp myBase;
        private Button btn;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);
                myBase = (BaseApp) getApplication();
                myBase.getObserver().addObserver(this);

                btn = (Button) findViewById(R.id.button1);
                btn.setText("value: " + myBase.getObserver().getValue());
                btn.setOnClickListener(this);

        }


        @Override
        public void update(Observable observable, Object data) {
                Toast.makeText(this, "I am notified" + myBase.getObserver().getValue(), 0).show();
                btn.setText("value: " + myBase.getObserver().getValue());
        }

        @Override
        public void onClick(View v) {
                startActivity(new Intent(ObjectObserverPattern.this,  SecondActivity.class));
        }
}
public class SecondActivity extends Activity implements Observer, OnClickListener {
   BaseApp myBase;
   private Button btn;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      myBase = (BaseApp) getApplication();
      myBase.getObserver().addObserver(this);
      btn = (Button) findViewById(R.id.button1);
      btn.setText("value: " + myBase.getObserver().getValue());
      btn.setOnClickListener(this);
   }

   @Override
   public void update(Observable observable, Object data) {
      // This method is notified after data changes.
      Toast.makeText(this, "I am notified" + myBase.getObserver().getValue(), 0).show();
      btn.setText("value: " + myBase.getObserver().getValue());
   }

   @Override
   public void onClick(View v) {
      myBase.getObserver().setValue("After Value Changed!");
   }
}
public class BaseApp extends Application {
   Test mTest;

   @Override
   public void onCreate() {
```

```
        super.onCreate();

        mTest = new Test();
    }

    public Test getObserver() {
        return mTest;
    }
}
public class Test extends Observable {
                private String name = "First time i have this Text";

                /**
                 * @return the value
                 */
                public String getValue() {
                                return name;
                }

                /**
                 * @param value
                 *          the value to set
                 */
                public void setValue(String name) {
                                this.name = name;
                                setChanged();
                                notifyObservers();
                }
}
```

3.6 Patrones de diseño. *Façade.*

El patrón *Façade* se aplicará cuando se necesite proporcionar una interfaz simple para un subsistema complejo, o cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel. Otro escenario proclive para su aplicación surge de la necesidad de desacoplar un sistema de sus clientes y de otros subsistemas, haciéndolo más independiente, portable y reutilizable (esto es, reduciendo dependencias entre los subsistemas y los clientes).

La principal ventaja del patrón *Façade* consiste en que para modificar las clases de los subsistemas, sólo hay que realizar cambios en la interfaz/fachada, y los clientes pueden permanecer ajenos a ello. Además, y como se mencionó anteriormente, los clientes no necesitan conocer las clases que hay tras dicha interfaz.

Como inconveniente, si se considera el caso de que varios clientes necesiten acceder a subconjuntos diferentes de la funcionalidad que provee el sistema,

podrían acabar usando sólo una pequeña parte de la fachada, por lo que sería conveniente utilizar varias fachadas más específicas en lugar de una única global.

```
public class Inventory {
    public String checkInventory(String OrderId) {
        return 'Inventory checked';
    }
}

public class Payment {
          public String deductPayment(String orderID) {
                    return 'Payment deducted successfully';

          }
}

public class OrderFacade {
          private Payment pymt = new Payment();
          private Inventory inventry = new Inventory();

          public void placeOrder(String orderId) {
                    String step1 = inventry.checkInventory(orderId);
                    String step2 = pymt.deductPayment(orderId);
          }
}

public class Client {
            public static void main(String args[]){
                      OrderFacade orderFacade = new OrderFacade();
                      orderFacade.placeOrder('OR123456');
            }
}
```

3.7 Obtención / Referencia de recursos

Como norma general todos los recursos que tengan un impacto significativo en el sistema, sobre todo en CPU y memoria, se obtendrán cuanto antes mediante el mecanismo habilitado para tal efecto.

Estos recursos se mantendrán disponibles y accesibles en la aplicación el tiempo estrictamente requerido y que dependerá del ciclo de vida de la aplicación y de las funcionalidades implementadas. El objetivo deberá ser siempre minimizar la instanciación e inicialización continuada de estos recursos, para optimizar los recursos utilizados y mejorar la experiencia del usuario al ser ésta mucho más fluida.

La inicialización de los recursos se debería realizar mientras se muestra la *splashscreen* de la aplicación, que es el punto habilitado para ello en los desarrollos de aplicaciones para dispositivos móviles, independientemente de su tecnología.

El acceso a estos recursos se deberá garantizar siempre a través del uso de patrones de diseño comunes como *Singleton*, *Factory*…

La liberación de los recursos deberá contemplar como mínimo el ciclo de vida de la aplicación. De esta manera e independientemente de la tecnología, la aplicación deberá liberar los recursos siempre que se encuentre en disposición de ser liberada de memoria ya sea por un evento del sistema operativo o por decisión del usuario.

En el caso de la aplicación pase a un segundo plano ***(****background****)*** ya sea por decisión del sistema (por ejemplo una llamada entrante) o por decisión del usuario, se deberá estudiar de manera individualizada para cada recurso si se ha de proceder a su liberación o si por el contrario se mantiene en memoria. En dicha decisión se deberá tener en cuenta el objetivo expuesto anteriormente.

La liberación de los recursos se deberá realizar siempre a través de la llamada al método habilitado para tal efecto y que dependerá del recurso en cuestión *(close(), dispose(), etc.).*

Una vez la aplicación vuelva a primer plano ***(****foreground****)*** los recursos liberados volverán a reservarse.

Los recursos que se deberían contemplar son:

- Input-Output stream
- Conexiones a base de datos, consultas y recuperación de datos de las mismas

- Conexiones a Content Providers, consultas y recuperación de datos de los mismos
- *Threads*
- Recursos gráficos ***(****drawables****)***
- Sockets
- Recursos físicos del dispositivo (Cámara, GPS…)

Como caso límite será obligatorio liberar todos los recursos de la aplicación si se produce en esta un error (o excepción) no controlada para, de esa manera, asegurar su disponibilidad en el sistema.

Para más información sobre los mecanismos existentes para la gestión de estos tipos de errores se deberá consultar el apartado de gestión de errores (o excepciones) para cada una de las tecnologías.

3.8 Constructores. No acceder método sobre-escribibles

Los constructores en ningún caso deben acceder a métodos sobre-escribibles. Solo se podrán acceder métodos ***private, static o final*** desde un constructor.

3.9 Constructores. Threads

Los constructores en ningún caso deben inicializar hilos de ejecución (Threads).

La inicialización de hilos de ejecución en un constructor puede provocar problemas con sub-clases y permite la referencia a *this* antes de que este haya sido construido completamente.

Esta limitación no se contempla en la instanciación (u obtención de referencia) del objeto que implementa el hilo de ejecución, siempre que la inicialización se realice en un método diferente al constructor.

3.10 Inicialización de campos a 0, false o *null*

Es obligatoria la inicialización de los campos a los siguientes valores:

- int, float, double: 0 o 0.0
- boolean: false
- referencias a objetos: null

3.11 Inicialización de Listas o Mapas

En ningún caso se deberá inicializar referencias a objetos de lista (List) o mapas (Map) a null.

Siempre se inicializarán usando los constructores por defecto y especificando el tipo (o tipos) soportados por la misma.

La referencia se deberá realizar siempre a través de las interfaces List y/o Map y nunca a través de la implementación de dichas interfaces.

De esta manera las siguientes inicializaciones serían consideradas incorrectas:

- List<String> miLista = null;
- List miLista = null;
- ArrayList miLista = null;

- Map<String, String> miMapa = null;
- Map miMapa = null;
- HashMap miMapa = null;

Mientras que las siguientes inicializaciones serían consideradas correctas:

- List<String> miLista = new ArrayList<String>();
- Map<String, String> miMapa = new HashMap<String, String>();

4 Estándar de construcción para aplicaciones Android

A continuación se detallan las consideraciones que deben tenerse en consideración en el desarrollo de aplicaciones Android.

4.1 Gestión de errores. Consideraciones generales para aplicaciones Android.

Los errores más comunes y más importantes que deben evitarse en un desarrollo para dispositivos móviles son, por orden de importancia para el usuario:

- Cierre inesperado de la aplicación (*crash*). Este error se produce siempre que se genera una excepción (error) no controlado en la aplicación. Dicho error es propagado por las diferentes capas de la aplicación hasta que se delega su gestión a la máquina virtual dalvik (DVM) que acaba finalizando la aplicación de manera abrupta. Normalmente este error puede llevar asociados errores de perdida de información del usuario y/o de corrupción de dicha información. Este error puede evitarse realizando una gestión de errores basado en capas y una propagación de errores controlada. A pesar de ello Android facilita un mecanismo para poder gestionar este tipo de errores en última instancia e impedir el cierre descontrolado de la aplicación (ver punto 4.2).

- Pérdida y/o corrupción de información. Estos errores suelen producirse cuando no se gestionan correctamente los estados del ciclo de vida de

la aplicación (paso a background, paso a foreground, cierre de la aplicación…) o cuando se realizan actualizaciones del modelo de datos de la aplicación en los procesos de *upgrade* de versión.

- *Application Not Responding* (ANR). Este tipo de error se produce cuando se realizan operaciones muy intensivas en procesado en el hilo principal de ejecución de la aplicación (UI-Thread). En esos casos se bloquea la respuesta de la vista (y sus componentes) y si esta situación dura más de 5 segundos, Android muestra un cuadro de diálogo de ANR que permite al usuario cerrar la aplicación o esperar a que esta recupere el estado inicial. Este error puede evitarse realizando una gestión correcta del uso de los Worker Threads en la aplicación (ver punto 4.3).

4.2 Gestión de errores. Captura de Excepciones incontroladas.

Cuando la aplicación se detiene inesperadamente y se cierra (*crash*) suele deberse en la mayoría de los casos a una excepción no controlada dentro del hilo principal de ejecución de la aplicación.

Para evitar que esto pase y que se pueda capturar la información del motivo del error que produce dicha excepción se deberá establecer un *handler* que será responsable de gestionar en última instancia todas las excepciones que se produzcan y que no sean gestionadas.

Para ello se deberá crear una clase que implemente la interfaz *UncaughtExceptionHandler*.

```
public class uncaughtExceptions implements UncaughtExceptionHandler{
        @Override
        public void uncaughtException(Thread thread, Throwable throwable) {
                // TODO Auto-generated method stub
        }
}
```

Dentro del método “uncaughtException” se podrá situar la implementación que gestionará la excepción y deberá realizar las siguientes acciones:

- Escribir reporte de error en la memoria externa del dispositivo (SDCard). Solo en modo *debug*.
- Escribir reporte de error a través de un servicio en el backend. A tal efecto se ofrecerá un servicio para el registro de errores en el ESB.
- Informar al usuario del error y ofrecerle la opción de inicializar la aplicación de manera automática y así evitar un efecto de crash.

El reporte de error debería incluir:

Información	Debug	Release
Versión de la aplicación	**X**	**X**
Nombre del paquete	**X**	**X**
Modelo del Terminal		**X**
Versión de Android	**X**	**X**
Versión del modelo		**X**
Marca	**X**	**X**
Dispositivo		**X**
Host		**X**
Identificador		**X**

Producto		**X**
Total memoria interna	**X**	**X**
Memoria interna disponible	**X**	**X**
Fecha y hora	**X**	**X**
Zona horaria		**X**

Esa clase se deberá asociar al Thread principal (o UI Thread) como *handler* por defecto en el método *onCreate()* de la *Activity* o de la clase que extienda de *Application,* mediante la siguiente implementación:

```
    Thread.setDefaultUncaughtExceptionHandler(new unCaughtExceptions());
```

4.3 Hilos de ejecución (Threads).

Un hilo de ejecución es una unidad concurrente de ejecución que puede ser llamada para realizar una o varias tareas en paralelo.

En Android existen dos maneras de ejecutar código fuente en un nuevo hilo:

- Se puede extender la clase *Thread* y sobre-escribir el método *run()* con la implementación que se quiera ejecutar en el nuevo thread (*Worker Thread*)

- Se puede instanciar un objeto *Thread* pasándole una clase que contenga la implementación que se quiere ejecutar en el nuevo *thread* (*Worker Thread*) y que implemente la interfaz *Runnable* como parámetro en el constructor.

En ambos casos es necesario llamar al método *start()* del objeto t*hread* para iniciar la ejecución del nuevo hilo de ejecución.

Cada hilo de ejecución tiene una prioridad que afecta a como el hilo será ejecutado por el sistema operativo. Un nuevo hilo hereda la prioridad de su padre y este valor también puede ser especificado usando el método *setPriority(int)* del objeto *thread.*

En una aplicación para dispositivos móviles basada en Android existe un hilo principal de ejecución (UI Thread - JVM) que es el responsable de la actualización de la vista, a través del ciclo de vida definido para las *Activities*, *Services* y para los *Fragments*.

Además es el responsable de la gestión de la pila de eventos de la aplicación ya sean estos eventos definidos por los elementos gráficos de la capa de presentación (elementos gráficos de control) o eventos derivados del sistema.

El bloqueo del UI Thread durante cinco (5) o más segundos produce que el sistema muestre un error de ANR (*Android Not Responding*) en forma de cuadro de diálogo que permite al usuario cerrar la aplicación.

Para evitar este error será obligatorio el uso de hilos de ejecución secundarios (*Workers Threads*) en los siguientes supuestos:

- **Operaciones de I/O.** Todas las operaciones de I/O con recursos locales (BBDD, Filesystem…) o con recursos externos (Servicios REST/JSON) se desarrollarán teniendo en cuenta una ejecución asíncrona a través del uso de un *Worker Thread*.

- **Operaciones con el Hardware.** Todos los accesos al hardware del dispositivo (Cámara, GPS, Acelerómetro…) y el acceso a *Content*

Provider publicados por otras aplicaciones se desarrollarán teniendo en cuenta una ejecución asíncrona a través del uso de un *Worker Thread.*

- **Funcionalidades intensivas en CPU.** Todas las funcionalidades que hagan un uso intensivo de la CPU (por ejemplo aplicación de filtros a imágenes) se desarrollarán teniendo en cuenta una ejecución asíncrona a través del uso de un *Worker Thread*.

4.4 Componentes de aplicación Android. Activities

Una *Activity* representa una vista con una interfaz de usuario y son por lo tanto los elementos principales de la capa de aplicación de una aplicación Android.

La *Activity* tiene un ciclo de vida que debe tenerse en cuenta para la correcta gestión de los estados de la aplicación (*background* / *foreground*). El ciclo de vida de una *Activity* es el siguiente:

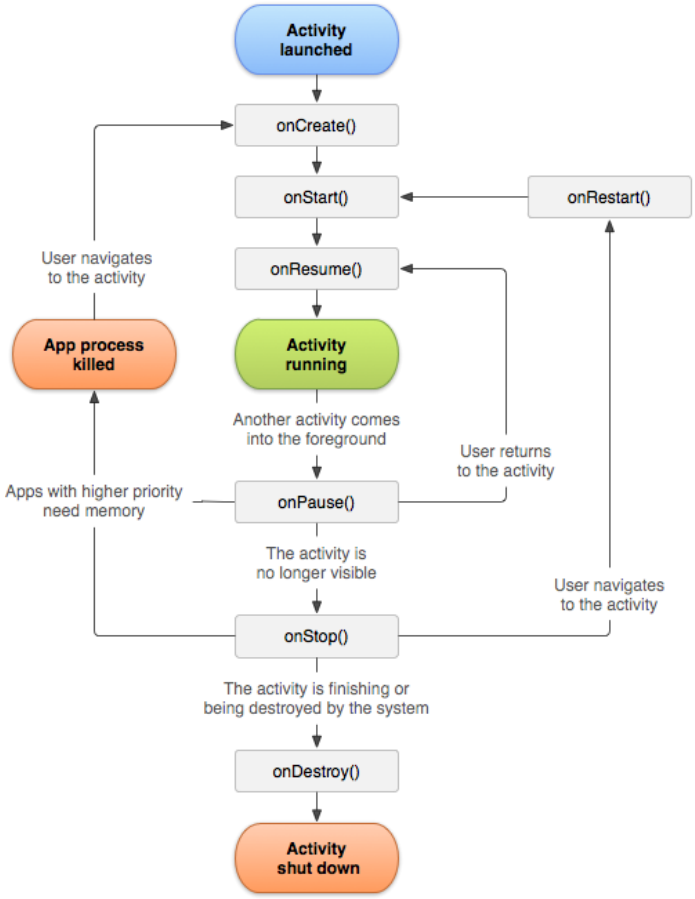


Ilustración 6: Ciclo de vida de una Activity

En la implementación de una *Activity* se deberá tener en cuenta los siguientes puntos:

- Sólo se declarará una *Activity* por fichero Java.
- El nombre de la *Activity* deberá seguir las reglas de estilo especificadas.

- Cada *Activity* deberá extender de la clase *Activity* o de la Clase *FragmentActivity***,** en función de si se hace uso de *Fragments* en la vista que representa la *Activity***.**

- Cada *Activity* se deberá encontrar declarada en el fichero de manifiesto de la aplicación.

Para todas las consideraciones de seguridad relativas al desarrollo de *Activities* consultar el punto 5.15.

4.5 Componentes de aplicación Android. Fragments.

Un *Fragment* representa una parte de la interfaz de usuario gestionada por una *Activity.*

Se pueden combinar múltiples Fragments en una misma Activity, y rehusarlos en múltiples *Activities*.

Cada *Fragment* tiene su propio ciclo de vida y está contenido dentro del ciclo de vida de la *Activity* que lo contiene. El ciclo de vida de un *Fragment* es:

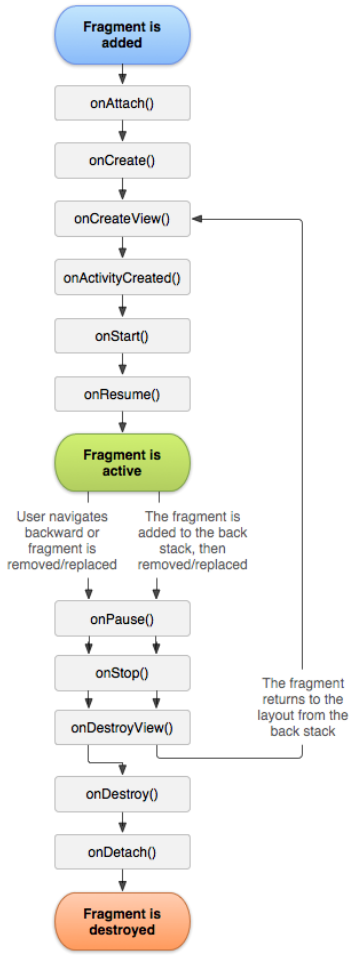


Ilustración 7: Ciclo de vida de un Fragment

En la implementación de *Fragments* se deberá tener en cuenta los siguientes puntos:

- Solo se declarará un *Fragment* por fichero Java.
- Cada *Fragment* deberá extender de la clase Fragment o bien de una subclase de ésta ***(****DialogFragment, ListFragment o PreferenceFragment****).***

- Los eventos de interfaz que afecten exclusivamente al fragment y no requieran comunicación con otro componente de interfaz, deberán ser manejados dentro del fragment en cuestión.

- Los eventos que requieran interacción con otros componentes de interfaz, se propagarán mediante el bus de eventos para que cualquier componente activo registrado a dicho evento sea capaz de reaccionar al evento.

- El nombre del *Fragment* deberá seguir las reglas de estilo especificadas.

Para todas las consideraciones de seguridad relativas al desarrollo de *Fragments* consultar el punto 5.15 (relativo a las consideraciones de seguridad en el desarrollo de *Activities*).

4.6 Componentes de aplicación Android. Services.

Un *Service* es un componente que se ejecuta en el UI Thread pero que no tiene representación gráfica. Esto es importante tenerlo en cuenta porque se tienen que tener las mismas consideraciones que en el caso de las *Activities* en cuanto a los casos de uso de los *Worker Threads* con el fin de evitar errores de ANR.

Los *Service* se deberán utilizar para operaciones de larga duración que deban ejecutarse sin interfaz gráfica y con un grado de prioridad menor que las *Activities.* Se usarán por lo tanto principalmente como interfaces de eventos del sistema, por ejemplo:

- Recepción y pre procesado de SMS específicos para la aplicación

- Recepción y pre procesado de llamadas telefónicas

- Recepción y pre procesado de Notificaciones PUSH

Los *Services* al igual que las *Activity* y los *Fragments* tienen un ciclo de vida que se tendrá que tener en cuenta en su implementación. El ciclo de vida de un *Service* es el siguiente:

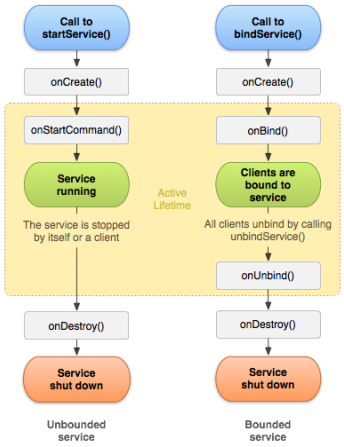


Ilustración 8: Ciclo de vida de un Service

En la implementación de *Services* se deberá tener en cuenta los siguientes puntos:

- Solo se declarará un *Service* por fichero Java

- Cada ***Service*** deberá reflejar en su implementación los métodos principales correspondientes a su ciclo de vida que se sobre-escriban. En el orden de definición de dichos métodos se deberá reflejar el orden de cada uno de ellos dentro del ciclo de vida especificado para el ***Service.*** De esta manera se deberán reflejar los siguientes métodos:

    1. onCreate()

    2. onStartCommand()**.** Si este método es implementado, es obligatorio parar el servicio una vez haya finalizado su tarea,

mediante la llamada a stopSelf() o stopService(). Si sólo se sobre-escribe onBind() esto no sería necesario.

3. onBind()**.** De implementación obligatoria. Este método se debe implementar cuando se quiere que otra aplicación pueda llamarlo. Debe proporcionar una interfaz para la comunicación del cliente con el servicio, mediante IBinder. Si no se quiere dar acceso exterior al servicio, se debe retornar null.

4. onDestroy(). De implementación obligatoria. Tendrá la responsabilidad de liberar todos los recursos reservados por el Service**.**

- Cada *Service* deberá extender la clase *Service* o *IntentService*, siendo esta última la recomendada si no es necesario que el *Service* maneje múltiples peticiones simultáneas.

- La comunicación entre un *Service* y las *Activities* de la aplicación se deberá realizar mediante el bus de eventos o la definición de un *Broadcast* y un *BroadcastReceiver* en función de las necesidades del proyecto*.* Si se usa un bus de eventos deberá definirse una clase para cada tipo de evento para evitar que los receptores reciban eventos en los que no están interesados*.*

- Cada *Service* se deberá encontrar declarado en el fichero Manifest de la aplicación.

- El nombre del *Service* deberá seguir las reglas de estilo especificadas.

Para todas las consideraciones de seguridad relativas al desarrollo de *Services* consultar el punto 5.18.

4.7 Componentes de aplicación Android. Content Providers.

Este componente permite compartir información entre aplicaciones que, en su modelo de ejecución basado en *sandbox*, se encuentran aisladas.

Por lo tanto se deberán implementar *Content Providers* en aquellas aplicaciones que requieran poder exponer una interfaz para poder compartir datos internos. Un ejemplo de aplicaciones que implementan *Content Provider* para compartir información se puede encontrar en la aplicación de Agenda Telefónica que incorpora Android.

En la implementación de *Content Providers* se deberá tener en cuenta los siguientes puntos:

- Sólo se declarará un *Content Provider* por fichero Java.
- Cada *Content Provider* deberá extender de la Clase *ContentProvider*.
- Cada *Content Provider* se deberá encontrar declarado en el fichero Manifest de la aplicación.
- El nombre del *Content Provider* deberá seguir las reglas de estilo especificadas.

Para todas las consideraciones de seguridad relativas al desarrollo de *Content Providers* consultar el punto 5.19.

4.8 Componentes de aplicación Android. Broadcast Receivers.

Un *Broadcast Receiver* es un componente que responde a un anuncio del sistema. Muchos *broadcast* originados por el sistema permiten que otras aplicaciones sepan que hay nueva información disponible para su uso, por ejemplo anunciando que la

batería está baja de carga, que ha sido tomada una foto, que la pantalla se ha apagado, etc. Una aplicación también puede iniciar un *broadcast* por ejemplo para hacer saber a otras aplicaciones que tiene nueva información para ellas.

A pesar de que un *broadcast* no lleva asociada una interfaz de usuario, éste puede crear notificaciones en la barra de estado, para de esta manera anunciar al usuario que un evento ha ocurrido. Este componente se considera una pasarela de información para otros componentes, por lo que se tiende a minimizar su tarea, y se centra en solo comunicar.

En la implementación de *Broadcast Receivers* se deberán tener en cuenta los siguientes puntos:

- Solo se declarará un *Broadcast Receiver* por fichero java.
- El nombre del *Broadcast Receiver* debe seguir las reglas de estilo especificadas.
- Cada *Broadcast Receiver* debe extender de la clase *BroadcastReceiver* y cada uno debe ser pasado como un *Intent.*
- Se debe implementar en el *Broadcast Receiver* su método principal:

    1. onRecieve()

- Cada *Broadcast Receiver* se deberá encontrar declarado en el fichero Manifest de la aplicación.

Para todas las consideraciones de seguridad relativas al desarrollo de *Broadcasts y Broadcasts Receivers* consultar el punto 5.16.

4.9 Componentes de aplicación Android. Intent.

Un *Intent* es un objeto que representa la voluntad de realizar una acción y es por lo tanto el mecanismo básico de comunicación en Android. Su uso común es para solicitar una acción a otro componente, y a través del cual se le puede pasar información.

Hay varias maneras en que un *Intent* se puede comunicar con otros componentes, pero las más comunes son:

- Para iniciar una *Activity***:** Se puede iniciar una nueva *Acitivity* pasando un *Intent* al método *startActivity().* Éste describe el tipo de *Activity* y puede contener información necesaria extra. Si se quiere recibir un resultado de otra *Activity*, cuando ésta finalice, se debe llamar al método *startActivityForResult()*. Nuestra *Activity* recibirá la información en un *Intent* nuevo mediante el método *callback onActivityResult().*

- Para iniciar un *Service***:** Se puede iniciar un *Service* para realizar una tarea pasando un *Intent* al método *startService()*. Este describe el servicio a iniciar y puede contener datos extras. Si el servicio es iniciado por una interfaz cliente-servidor, se puede enlazar (*bind*) el servicio desde otro componente, pasando un *Intent* al método *bindService().*

- Para entregar información a un *Broadcast***:** se puede lanzar un *broadcast* para otras aplicaciones pasando un *Intent* al método *sendBroadcast(), sendOrderedBroadcast()* o *sendStickyBroadcast(),* el cual contendrá la información a entregar.

La información básica que contiene un *Intent* es:

- *Component name:* representa el nombre del componente a inicializar. Se le debe pasar explícitamente, definiéndolo en el *Manifest* preferiblemente, de lo contrario el sistema asume que es implícito y decide que componente recibe el *Intent* basado en el resto de la información.

- *Action:* especifica una acción genérica a realizar, como puede ser: View, Pick, etc.

- *Data:* es un objeto tipo *Uri* que representa la información o el tipo de MIME. Este suele estar asociado con la acción a realizar. Ejemplo, si la acción es *ACTION_EDIT*, este campo debe contener la *URI* del documento a modificar.

- *Category:* contiene información adicional acerca del tipo de componente que debe manejar el *Intent.*

- *Extras:* información adicional, en forma de *key-value pair*, que contiene los datos necesarios para realizar la acción, a través del método *putExtra()*, se puede crear un objeto *Bundle* con todos los datos extras y pasarlo al método *putExtras()*. Ejemplo, si se quiere poder dar la opción al usuario de enviar un correo, en los extras se debe pasar la dirección email a la cual el usuario escribirá, también se puede pasar el asunto, y parte del cuerpo del mensaje.

En la implementación de los *Intent* se deberán tener en cuenta los siguientes puntos:

- Cada *Activity* que requiera manejar información a través de *Intent*, estos deben ser declarados en el *Manifest*, en la etiqueta correspondiente a su *Activity:*

```
    <activity android:name="ShareActivity" >
        <!-- This activity handles "SEND" actions with text data -->
        <intent-filter>
            <action android:name="android.intent.action.SEND" />

            <category android:name="android.intent.category.DEFAULT" />

            <data android:mimeType="text/plain" />
        </intent-filter>
    </activity>
```

- El nombre del *Intent* debe seguir las reglas de estilo especificadas.
- Se deben especificar explícitamente la información básica asociada a los *Intent* antes mencionada.

Para todas las consideraciones de seguridad relativas al desarrollo de *Intent* consultar el punto 5.14.

4.10 Componentes de aplicación Android. Adapters.

Android usa *Adapters* como puente de comunicación entre componentes de la UI y la información que será asociada a este componente. Este brinda acceso a cada elemento de los datos, y es responsable de construir una vista para cada elemento.

Existen varios tipos de *Adapter*, y pueden ser consultados en este enlace. Pero quizás el de más uso es el *BaseAdapter*, por ejemplo, para llenar la información de un *ListView*, y personalizar cada elemento de forma automática.

En la implementación de un *Adapter* se deberá tener en cuenta los siguientes puntos:

- Solo se declarará un *Adapter* por fichero Java.
- Siempre que se haga uso de un *Adapter*, se debe crear una clase que extienda de *Adapter*.
- El nombre de la clase debe seguir las reglas de estilo especificadas.

- Se deberá crear una clase *Holder* para manejar los componentes de la vista con un mejor rendimiento.

Ejemplo complete: En la clase donde se obtiene la referencia al componente *ListView*, se declara el adaptador y se le asigna al *ListView*.

```
public class myActivity extends Activity {
        private ArrayList<ItemPersonalized> mItems;
        private ListViewAdapter adapter;
        private ListView list;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                //components
                list = (ListView) findViewById(R.id.list);

                //set adapter
                adapter = new ListViewAdapter(mItems, null);
                list.setAdapter(adapter);
        }
        ...
}
```

La clase *ItemPersonalized* representa el modelo de datos de cada elemento de la lista. La Clase *ListViewAdapter* es el adaptador personalizado, el cual extiende de *BaseAdapter.*

```
public class ItemPersonalized {
          private String mLogo;
          private String mName;
          private String mDescription;
          public ItemPersonalized() {
      …
          }

          public ItemPersonalized(String mLogo, String mName, String mDescription)     {
                    super();
                    this.mLogo = mLogo;
                    this.mName = mName;
                    this.mDescription = mDescription;
          }
          public String getmLogo() {
                    return mLogo;
          }
          public void setmLogo(String mLogo) {
                    this.mLogo = mLogo;
          }
          public String getmName() {
                    return mName;
          }
          public void setmName(String mName) {
                    this.mName = mName;
          }
          public String getmDescription() {
                    return mDescription;
          }
          public void setmDescription(String mDescription) {
                    this.mDescription = mDescription;
          }
}

public class ListViewAdapter extends BaseAdapter {
          private ArrayList<ItemPersonalized> items;
          private OnClickListener onClick;
          /** Constructors filling just items*/
          public ListViewAdapter(ArrayList<ItemPersonalized> it) {
                    this.items = it;
          }
          /** Constructors filling items and each item click behaivour*/
          public ListViewAdapter(ArrayList<ItemPersonalized> it, OnClickListener cl) {
                    this.items = it;
                    this.onClick = cl;
          }

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

          @Override
          public Object getItem(int arg0) {
                    return items.get(arg0);
          }
          @Override
          public long getItemId(int position) {
                    return position;
          }
          /** Personalizing View for each item */
          @Override
          public View getView(int position, View view, ViewGroup parent) {
                    view_holder row_view;
                    if (convertView == null || convertView.getTag() == null) {
                              convertView = inflat-er.inflate(R.layout.row_item_personalized, null);
                              row_view = new view_holder();
                              row_view.name = (TextView) con-vertView.findViewById(R.id.tv name);
                              row_view.desc = (TextView) con-vertView.findViewById(R.id.tv desc);
                              row_view.logo = (ImageView) con-vertView.findViewById(R.id.tv_name);
                              row_view.ll_holder              =              (LinearLayout)              con-
vertView.findViewById(R.id.ll_holder);
```

```
                        convertView.setTag(row_view);
                } else {
                        row_view = (view_holder) convertView.getTag();
                    }
                // fill data
                ItemPersonalized it = items.get(position);

                row_view.name.setText(it.getmName());
                row_view.desc.setText(it.getmDescription());
                row_view.logo.setImageResource(it.getmLogo());
                row_view.ll_holder.setTag(position);
                if (onClick != null)
                        row_view.ll_holder.setOnClickListener(onClick);

                convertView.setTag(row_view);
                return convertView;
        }


        /** View holder class is recommended*/
        private class view_holder {
                public TextView name;
                public TextView desc;
                public ImageView logo;
                public LinearLayout ll_holder;
        }
}
```

Siendo *row_item_personalized* el XML que define cada elemento, el cual se ha personalizado usando componentes tipo *TextView* e *ImageView*, en este caso.

4.11 Componentes de aplicación Android. AsyncTask.

Los *AsyncTask* son componentes de Android que permiten la ejecución de lógica en un hilo de ejecución secundario (*Worker Thread*) al mismo tiempo que permite a través de la implementación de algunos de sus métodos de la ejecución de cierto código en el UI Thread de la aplicación.

En la implementación de un *AsyncTask* se deberá tener en cuenta los siguientes puntos:

- Solo se declarará una *AsyncTask* por fichero Java.
- El nombre de la *AsyncTask* debe seguir las reglas de estilo.
- Cada *AsyncTask* debe extender de la clase *AsyncTask<String, X…>.*

4.12 Componentes de aplicación Android. Persistencia.

Android brinda diferentes opciones para la persistencia de la información de nuestra aplicación. Su uso está condicionado a las necesidades específicas de la aplicación.

Pautas a seguir para su uso e implementación:

- ***Shared Preference***: Se usa para almacenar datos primitivos en *key-values pair*.
    - Usar una única *Shared Preference* por *aplicación.*
    - El nombre de la *Shared Preference* debe seguir las reglas de estilo especificadas.
    - Se deberá usar de manera racional, ya que aumenta el espacio de la aplicación y su operativa es costosa en ciclos de CPU. Hay que tener en cuenta que este tipo de persistencia hace uso del *Filesystem* de la aplicación.
- ***Internal Storage:*** Se usa para almacenar datos privados en la memoria interna del dispositivo.
    - Su declaración debe seguir las reglas de estilo especificadas.
    - Si se usa, usar siempre MODE_PRIVATE, salvo expresa especificación por necesidad de compartir la información con otras aplicaciones.
    - Se deberá usar de manera racional, ya que aumenta el espacio de la aplicación y su operativa es costosa en ciclos de CPU. Hay que tener en cuenta que este tipo de persistencia hace uso del *Filesystem* de la aplicación.

- ***External Storage:*** Se usa para almacenar datos de carácter público en la memoria externa del dispositivo
    - Su declaración debe seguir las reglas de estilo especificadas.
    - Se debe declarar en el *Manifest*, el permiso asociado a esta opción, para aprobación del usuario.
    - Antes de su uso se debe comprobar si está disponible, y si está disponible para lectura, escritura o ambos.
- ***SQLite storage:*** Se usa para almacenar información estructural y relacionada en una base de datos.
    - Para cada base de datos, se creará una clase que extienda de SQLiteOpenHelper.
    - Cada clase se declarará en un fichero Java.
    - Su declaración debe seguir las reglas de estilo especificadas.
    - Todas las operaciones relacionadas con la creación, acceso y consulta a la base de datos se hará a través de un hilo secundario o de un AsyncTask en los casos en los que haya que reportar el avance de la tara al hilo principal.
- ***Networks Connections:*** Se usa para almacenar información servidor a través de conexiones a internet usando servicios web.
    - El manejo y tratamiento de estas conexiones se hará a través de una única clase, implementada en un fichero Java.
    - Su declaración debe seguir las reglas de estilo especificadas.

- Todas las operaciones relacionadas con conexión, envío y recepción de información de la red se hará a través de un hilo secundario (la librería Volley permite hacerlo) o de un AsyncTask en los casos en los que haya que reportar el avance de la tara al hilo principal.

- Esta clase no debe procesar la información que se envía al servidor ni la información recibida desde éste. Su responsabilidad debe ser exclusivamente la de gestionar la conexión y la transmisión de la información.

- Para su implementación se debe usar URLConnection.

- Los servicios web asociados, del lado del servidor, se deben implementar usando REST/JSON.

- Para realizar los procesos de serialización/deserialización de la información enviada o recibida por un servicio se recomienda el uso de la librería *Gson* (https://code.google.com/p/google-gson/)

4.13 Componentes de aplicación Android. Logs.

Para la depuración y seguimiento de nuestra aplicación, estando en modo *debug*, se usará la API de *logging* que Android proporciona.

Pautas generales a seguir:

- Solo será visible cuando se esté en desarrollo y con la opción *debug* activada. Esta opción puede ser controlada por una variable, a nuestra conveniencia.

- El *TAG* definido para cada log se nombrará con el nombre de la clase y el método de la misma en la cual se esté registrando el *log.*
- El mensaje del *log* no debe ser excesivamente largo, debe ser claro y auto-suficiente. Su contenido debe ir en pares preferiblemente (*key-values pair*).
- La clasificación de cada *log* va a estar directamente relacionada con la criticidad del log que se quiere generar, respetando:

Criticidad	Clasificación
Error	e()
Warning	w()
Info	i()
Debug	d()
Verbose	v()

El registro de *Logs*, debe ser clasificado en tres grandes grupos:

- Eventos del sistema
- Actividad
- Debug

4.14 Componentes de aplicación Android. Internacionalización.

Android permite el manejo de texto, audio y gráficos basado en el criterio de internacionalización (incluso cuando la aplicación se esté desarrollando para un solo

idioma) a través del uso de calificadores sobre los recursos definidos en la aplicación.

Para realizar una correcta implementación de la internacionalización se deberán tener en cuenta los siguientes puntos:

- Todo el texto en los *layout* deben ser definido haciendo uso del recurso @string.
- Todo el texto a mostrar al usuario, utilizado en la parte Java, debe hacer referencia a su declaración en el recurso @string, y acceder a él programáticamente.
- El mismo criterio se aplica para archivos de audio y/o gráficos personalizados, que contengan, por ejemplo, texto o referencias regionales.

4.15 Vistas en Android. Accesibilidad.

La interacción con dispositivos Android se puede dar de diversas maneras. Los usuarios que tienen limitaciones visuales, físicas o relacionadas con la edad pueden tener problemas para definir o usar correctamente una pantalla táctil, y usuarios con problemas auditivos quizás no sean capaces de percibir información audible y alertas.

Para este tipo de usuarios, Android brinda características y servicios de accesibilidad para ayudar a que la interacción con los dispositivos sea más fácil, incluyendo *text-to-speech,* navegación mediante gestos*, trackball* y navegación *D-pad*. Si se desea que la aplicación esté disponible para este tipo de público, se debe tener en cuenta estos servicios y usarlos a conveniencia.

Pautas a seguir para realizar aplicaciones accesibles para todo tipo de usuarios:

- Proporcionar texto descriptivo para todos los controles de la interfaz usando el atributo *android:contentDescription*. Prestar particular atención a *ImageButton*, *ImageView* y *CheckBox*.
- Asegurarse de que todos los elementos de la interfaz que aceptan entradas son accesibles por los controles direccionales.
- Asegurarse de que todo el audio que se muestra al usuario va acompañado de información visual o una notificación.
- Probar la aplicación usando solo servicios de navegación de accesibilidad alternativos.

4.16 Vistas en Android. Temas y estilos.

Un estilo es una colección de propiedades que especifican un formato y apariencia a una *View* o a una ventana. Puede definir propiedades tipo *height*, *padding*, *font color*, *font size*, *background color* y mucho más. Se definen en un recurso *XML* separado del *XML* que define el *layout*.

El uso de estilos es recomendable para:

- Hacer los *layout* más entendibles y limpios.
- Reutilizar estilos en varios *layouts.*
- Hacer más modular el diseño de la interfaz.

Existen dos maneras de aplicar temas y estilos para la UI:

- A un componente especifico, a través del atributo *style* en el *layout.*
- A toda una *Activity* o a la aplicación, mediante el atributo *android:theme* del elemento *<activity>* o *<application>* en el *Manifest*.

Ejemplo añadiendo un estilo a un componente especifico. Este código de ejemplo muestra un *layout* tradicional sin usar estilo:

```
<TextView android:layout_width="fill_parent" android:layout_height="wrap_content"
android:textColor="#00FF00" android:typeface="monospace" android:text="@string/hello" />
```

Y el mismo *layout* usando un estilo:

```
<TextView style="@style/CodeFont" android:text="@string/hello" />
```

Para este ejemplo, el estilo *CodeFont* está definido en un XML de la siguiente manera:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CodeFont" parent="@android:style/TextAppearance.Medium">
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

Es preceptivo por lo tanto el uso de estilos para definir la apariencia de las vistas.

4.17 Vistas en Android. Personalización de componentes visuales.

Una lista parcial de algunos de estos componentes incluye: *Button, TextView,EditText, ListView, CheckBox, RadioButton, Gallery, Spinner*, y para un propósito más específico *AutoCompleteTextView, ImageSwitcher, y TextSwitcher*. Además de los *layouts* disponibles *LinearLayout, FrameLayout, RelativeLayout* y otros.

Si ninguno de estos componentes o *layouts* satisfice las necesidades del proyecto, se puede crear uno nuevo personalizado extendiendo la clase *View*. Si solo se desea realizar pequeños ajustes, se puede extender de la clase componente y sobre-escribir sus métodos.

La recomendación es usar siempre los componentes básicos de Android y personalizarlos sin necesidad de crear uno nuevo, por cuestión de rendimiento y compatibilidad. La creación de nuevos componentes sólo se realizará cuando sea estrictamente necesario y bajo aprobación del departamento de Arquitectura.

4.18 Estructura de proyecto Android

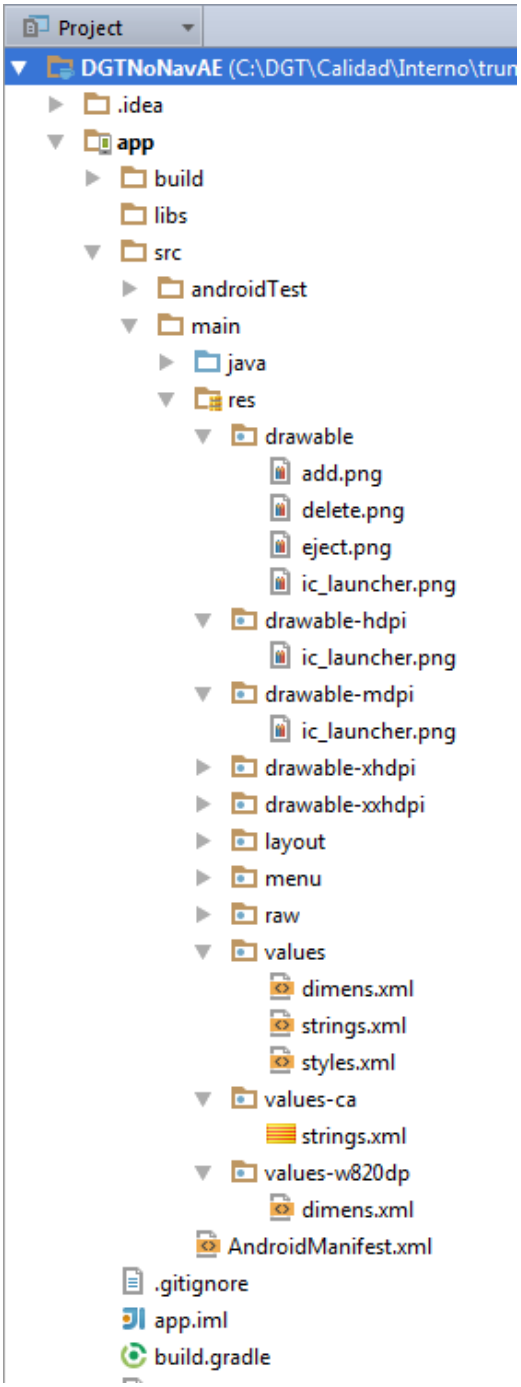


Ilustración 9: Estructura de proyecto Android

- **AndroidManifest.xml:** Este fichero describe la aplicación Android. En él se indican las *Activities*, *Intents*, *Services* y los *Content Providers* de la aplicación. También se declaran los permisos que requerirá la aplicación, la versión mínima de Android para poder ejecutarla, el paquete Java, la versión de la aplicación, etc.

- **/src/androidTest:** Carpeta que contiene el código fuente destinado a insertar código de test de la aplicación utilizando el API JUnit

- **/src/main/java:** Carpeta que contiene el código fuente de la aplicación. En el proyecto ejemplo de la ilustración podemos ver:

- **/src/main/res:** Carpeta que contiene los recursos usados por la aplicación.

    - **drawable**: En esta carpeta se almacenan los ficheros de imágenes (JPG o PNG) y descriptores de imágenes en XML. Podrán existir directorios adicionales con el identificador de densidad gráfica de los recursos que contiene (drawable-hdpi, drawable-mdpi…) En el proyecto ejemplo de la ilustración, podemos ver el fichero ic_launcher.png que será utilizado como icono de la aplicación y se ha añadido en cuatro versiones diferentes, de forma que solo se cargue al cumplirse una determinada condición. *Por ejemplo: los recursos contenidos en la carpeta drawable-hdpi se cargarán cuando el dispositivo donde se instala la aplicación tiene una densidad gráfica alta (~240 dpi);*

    - **layout**: Contiene ficheros XML con las vistas de la aplicación.

    - **menu**: Contiene ficheros XML con los menús de cada actividad.

- **values**: Contiene ficheros XML para indicar valores usados en la aplicación, que nos permitirá cambiarlos desde estos ficheros sin necesidad de ir al código fuente. Los ficheros principales que tendremos en esta carpeta son las siguientes:

  - En **dimens.xml**, se definen las dimensiones (por ejemplo el margen horizontal y vertical). En el proyecto ejemplo, podemos ver dimens.xml en 2 directorios: *"values"*, el utilizado por defecto y *"values-w820dp"*, que será utilizado en dispositivos con ancho superior a 820 dp (esto ocurrirá en tabletas).

  - En el fichero **strings.xml**, se definen todas las cadenas de caracteres de la aplicación. En los casos en que sea necesario que el texto de la aplicación se muestre en más de un idioma se generarán directorios adicionales con el identificador de idioma (values-en, values-es…) que contendrán un fichero strings.xml distinto por cada idioma

  - En el fichero **styles.xml**, se definen los estilos y temas de la aplicación.

- **xml**: Contiene otros ficheros XML requeridos por la aplicación.

- **raw**: Contiene ficheros adicionales que no se encuentran en formato XML.

- **Gradle Scripts:** Ficheros gradle que permiten construir la aplicación.

5 Seguridad en Android

La seguridad en las aplicaciones para dispositivos móviles hereda muchos de los retos asociados a la seguridad Web (conectividad continuada – aunque fluctuante -, amplia audiencia…) combinados con aquellos riesgos de seguridad asociados al desarrollo de aplicaciones tradicionales (cliente/servidor) como son: gestión de buffers, encriptación local, etc.

A nivel de seguridad se deberán tener en cuenta los siguientes actores:

- Device Layer
- Network Layer
- Data Center

Se deberá tener en cuenta que los ataques pueden tener como origen una de estos actores o una combinación de varios.

A continuación se enumeran las directrices que serán de obligado cumplimiento en todos los desarrollos Android de la DGT. Todas aquellas excepciones aplicadas a lo aquí indicado deberán contar con la autorización del Departamento de Arquitectura.

También se indica el grado de relevancia de cada una de las directrices en función del ámbito tecnológico de la aplicación (Nativa, Híbrida o Web móvil) y proporcionará el grado de gravedad en caso de incumplimiento (Crítico, Medio, Bajo).

5.1 *Device Layer*. Almacenamiento local de información

Nativo	Híbrido	Web móvil
Crítico	Crítico	Bajo

Independientemente del tipo de información (imágenes, ficheros de texto…), de su naturaleza (confidencial o pública) y seguridad aplicada (encriptada o en texto plano) toda la información que se almacene localmente puede ser comprometida.

Hay que tener en cuenta que debido a la gestión agresiva que se realiza en las memorias NAND Flash para preservar su durabilidad, la información almacenada en las mismas no es eliminada físicamente cuando se realiza dicha acción, sino que los espacios de memoria asociados se marcan como disponibles. Lo que permitiría a un atacante poder recuperar toda aquella información en un principio eliminada. Además dicha política impide el borrado mediante sobre-escritura (de sectores) propio de otros medios de almacenamiento físico como son, por ejemplo, los HDD.

Teniendo todo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Siempre que sea posible no debe almacenarse información en el dispositivo, con un especial énfasis en aquella información que sea confidencial o sensible.
- Siempre que sea posible se mostrará información y se transmitirá, pero no se persistirá en la memoria RAM del dispositivo o en la memoria NAND Flash.
- Si ha de almacenarse información en el dispositivo se deberá tener una preferencia por la memoria RAM sobre la memoria NAND Flash.

- Si ha de almacenarse información confidencial o sensible esta deberá encontrarse convenientemente cifrada mediante algoritmos de cifrado fuerte.

- Siempre que sea posible no se deberán almacenar localmente información relacionada con *passwords* ni claves maestras. En aplicaciones Android, siempre que deba almacenarse información confidencial o sensible se deberá realizar en el Internal Storage de la aplicación y se deberá utilizar el mecanismo proporcionado por Android a través del uso del método *setStorageEncription* de la clase *DevicePolicyManager*. Este mecanismo proporciona encriptación a del Internal Storage de la aplicación. Cuando se deba almacenar información confidencial o sensible en otros almacenamientos (por ejemplo SDCard) se deberá validar siempre el *flag* "isExternalStorageEmulated()". Si el valor del flag es true este almacenamiento heredará la política de seguridad y encriptación proporcionada por el uso del método *setStorageEncription*. En caso contrario (valor del *flag* igual a *false*) se deberá utilizar otro mecanismo que asegure el cifrado de la información almacenada en el External Storage (ver punto siguiente).

- El cifrado de información confidencial o sensible deberá basarse en el uso de una *master key* que a su vez deberá estar protegida mediante un *password*, usando una *key derivation function*. En el caso de Android esa *key derivation function* deberá ser PBKDF2WithHmacSHA1. Tanto la *master key* como la *password* se deberán almacenar siempre en el Internal Storage codificadas mediante los mecanismos definidos por el uso del método

setStorageEncription de la clase *DevicePolicyManager* (ver punto anterior).

- Si han de almacenarse capturas de pantalla de la aplicación, se deberá tener en cuenta que en dichas capturas no aparezca ninguna información confidencial o sensible.

- El departamento de Arquitectura podrá en cualquier caso modificar o ampliar las presentes directrices en función de las capacidades técnicas proporcionadas por la solución de MDM homologada en la DGT.

5.2 *Device Layer*. Captura local de información

Nativo	Híbrido	Web móvil
Crítico	Crítico	Bajo

Relacionado con el punto anterior hay multitud de información que se captura o se genera en una aplicación: *logs*/ficheros de depuración, *cookies*, historial web, caché de navegación, listas de propiedades, ficheros y bases de datos SQLite.

Teniendo todo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Siempre que sea posible no deberá almacenarse en el dispositivo la información anteriormente enumerada.

- Si ha de almacenarse información sobre *logs*, no deberá recogerse ningún tipo de información sobre los *crashes* que puedan ocurrir para,

de ese modo, no dar pistas a un potencial atacante sobre vulnerabilidades de la aplicación.

- Si ha de almacenarse información sobre logs, estos sólo estarán disponibles en las *releases* de desarrollo. En ningún caso en las releases de producción.

- Si han de almacenarse ficheros, se deberán seguir las directivas establecidas en el punto 5.1.

- Siempre que sea posible se emplearán las directivas establecidas en la documentación de la plataforma para evitar la caché en las comunicaciones HTTP (En el SDK de Android, *android.net.http*).

- Si no es posible evitar el cacheado de datos a través de la implementación de la librería de comunicaciones, se deberá optar siempre por una estrategia alternativa. Por ejemplo, se puede concatenar en la URL un parámetro (*unused*) que adoptará un valor diferente para cada petición (por ejemplo, el *timestamp* del sistema).

- Siempre que sea posible se deberá utilizar en el web server la directiva “no-store” para evitar el cache de información relativa a la respuesta HTTP en el navegador.

5.3 *Device Layer*. Envío de información sensible o confidencial

Nativo	Híbrido	Web móvil
Crítico	Crítico	Crítico

Los parámetros de una petición (por ejemplo en peticiones HTTP del tipo GET/DELETE) son visibles y son susceptibles de ser colocados en memorias caché (historial de navegación, *logs* del servidor web, *logs* del proxy, etc.).

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Siempre que sea posible se deberá evitar enviar información sensible o confidencial como parámetro de la petición. En su lugar, dicha información deberá ubicarse en el cuerpo de la petición y deberá encontrarse correctamente cifrada y firmada.

- Siempre que sea posible se deberá incluir dentro del cuerpo el uso de un *token* para evitar ataques del tipo CSRF (*Cross-Site Request Forgery*). Este *token* puede gestionarse mediante un patrón del tipo *Synchronizer Token* y puede encontrarse más información sobre su implementación en:

  https://www.owasp.org/index.php/CrossSite_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

- Si ha de enviarse información sensible o confidencial como parámetro de la petición, esta deberá encontrarse correctamente cifrada.

5.4 *Device Layer*. Evitar la detención inesperada de la aplicación (Crash).

Nativo	Híbrido	Web móvil
Crítico	Crítico	Bajo

Una de las fuentes principales de información que pueden ayudar a un atacante a sobrepasar la seguridad implementada en una aplicación son aquellos errores que producen la detención inesperada de la misma.

Hay que tener en cuenta que cada error de este tipo que se produce a parte de indicar al atacante la existencia de una debilidad en la programación de la aplicación, también proporciona información valiosa a través de los *logs* sobre como reproducirla (y cómo poder explotarla).

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Durante el proceso de desarrollo de la aplicación se aplicará un estilo de programación defensivo (validación de parámetros, etc.) y una gestión de errores basados en las diferentes capas de la aplicación con una propagación controlada de los mismos.

- Será obligatorio establecer un mecanismo programático que impida que la aplicación se pare de manera inesperada en el caso de que se produzca un error no controlado. Para ello ver las directrices recogidas en el punto 4.2.

- En las releases de producción no deberá quedar ningún tipo de constancia ni información asociada al error en los logs de la aplicación o del sistema. Logs de la aplicación que, siguiendo las directrices recogidas en el punto 5.2, deberían encontrarse deshabilitados.

- En el *feedback* proporcionado al usuario a través de la interfaz se deberá utilizar una redacción que evite a un potencial atacante deducir la existencia de este tipo de errores. De esta manera se deberán evitar

redacciones conteniendo, por ejemplo, las siguientes expresiones: (Error) Fatal, (Error) Inesperado, Reinicio (de aplicación), etc.

5.5 *Device Layer*. Caducidad de sesiones locales

Nativo	Híbrido	Web móvil
Medio	Medio	Bajo

Un dispositivo perdido o robado puede permitir a un atacante realizar operaciones de alto riesgo (acceder a datos sensibles o confidenciales, ejecutar transacciones…). Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Es un requerimiento imprescindible que aquellas aplicaciones que utilicen datos sensibles o que realicen transacciones de alto impacto para el negocio (aquellas que pueden representar pérdidas de dinero, pérdidas de imagen, pérdidas de clientes…) establezcan un sistema de sesiones locales que será responsable de establecer una política de accesos basada en tiempo.

- En ningún caso las sesiones locales deberán perdurar activas sin ningún tipo de actividad más del tiempo establecido según el nivel de seguridad de la aplicación:

    - 15 minutos para nivel alto

    - 30 minutos para nivel medio

    - 60 minutos para nivel bajo

- Cuando una sesión local caduca por inactividad deberá requerir la autenticación del usuario para poder acceder de nuevo a la funcionalidad de la aplicación.

- Cuando una sesión local caduca por inactividad deberá eliminar toda la información relativa a los datos del usuario y de aquellas credenciales *(master keys)* que se estén utilizado para el cifrado/descifrado de información local.

5.6 *Device Layer*. Ofuscación de datos sensibles en la interfaz gráfica.

Nativo	Híbrido	Web móvil
Medio	Medio	Medio

La presentación en la interfaz de datos sensibles o confidenciales (números de cuentas bancarias, PIN, passwords…) puede representar un riesgo importante de seguridad. Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Siempre que sea posible se deberá evitar mostrar información sensible o confidencial en la interfaz gráfica.

- Si ha de mostrarse información sensible o confidencial en la interfaz gráfica, ésta se mostrará ofuscada de manera que no se muestre completa sino sólo aquella parte de la misma que permita, por ejemplo, su validación por parte del usuario.

- Si ha de mostrarse información sensible o confidencial en la interfaz gráfica de manera clara y completa, se deberá pedir siempre la

autorización del usuario avisando de manera clara y concisa de los riesgos de seguridad que supone y de las medidas que debería adoptar para mitigarlos.

- Siempre que sea posible se deberá trabajar en la aplicación móvil con elementos que puedan identificar la información sensible requerida de manera unívoca pero que no permita a un atacante derivar la información sensible a partir de dichos elementos (por ejemplo, usando un *token* único por sesión que permita asociar en el backend dicho token con una cuenta corriente).

- En procesos de autenticación siempre se deberá devolver el mismo mensaje de error, independientemente de que el error de validación se produzca sobre el campo usuario o el campo password. El mensaje de error deberá tener una redacción neutra de manera que no facilite información a un potencial atacante.

5.7 *Device Layer*. Ofuscación de código.

Nativo	*Híbrido*	Web móvil
Medio	*Medio*	Bajo

Cuanta mayor sea la complejidad del código y más compleja la posibilidad de realizar ingeniería inversa de la aplicación, mayor será el nivel de seguridad alcanzado a ataques de substitución o inyección de código. Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Todas las aplicaciones Android deberán presentar una releases de producción ofuscadas según las directrices recogidas en la documentación de referencia de desarrollo de Android:

http://developer.android.com/google/play/billing/billing_best_practices.html#obfuscate

http://developer.android.com/tools/help/proguard.html

5.8 *Device Layer*. Restricción de depuración.

Nativo	Híbrido	Web móvil
Crítico	Medio	Bajo

La posibilidad de conectar un depurador al proceso de ejecución de la aplicación es un riesgo de seguridad que puede ser aprovechado por un atacante para conseguir información de bajo nivel del entorno de ejecución de la aplicación (*runtime*) como paso previo a ataques de este nivel.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- En aplicaciones Android, se deberá configurar el proceso de automatización (*gradle*) para que todas las releases que puedan ser distribuibles fuera del ámbito del equipo técnico de la aplicación, sean éstas de desarrollo o de producción, contengan en el manifiesto de la aplicación la directiva:

android:debuggable="false"

El objetivo es minimizar el riesgo de distribución incontrolada de una versión que pueda ser depurada.

- Se deberá valorar la implementación a través del uso del NDK de Android (a través de JNI – Java Native Interfaces -) todos aquellos procesos que sean especialmente sensibles a análisis o ataques basados en la depuración. Son especialmente susceptibles de este tipo de implementación aquellos que requieran una gran carga computacional o lógica completa, ya que aparte de seguridad contra la depuración este tipo de implementación aporta también una mejora substancial en el rendimiento.

5.9 *Device Layer*. Caché de teclados.

Nativo	Híbrido	Web móvil
Medio	Medio	Medio

Android contiene un diccionario asociado al usuario de manera que palabras introducidas por el usuario se guardan para poder ser mostradas como sugerencias en futuros usos. Este diccionario asociado al usuario se encuentra disponible para cualquier aplicación sin necesidad de proporcionar privilegios adicionales.

- Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices: Deberá valorarse junto al departamento de Arquitectura, la opción de desarrollar un teclado particular para la aplicación o utilizar el teclado corporativo, siempre que existan funcionalidades que requieran la interacción del usuario con información sensible o confidencial a través de este tipo de

componente gráfico. En dicho teclado se deberá asegurar la desactivación del diccionario asociado al usuario.

5.10 *Device Layer*. Copy & Paste.

Nativo	Híbrido	Web móvil
Medio	Medio	Medio

Android por defecto incorpora soporte para las funcionalidades de copy & paste sin restricciones y sin necesidad de proporcionar privilegios adicionales. Esto representa un riesgo seguridad ya que permite a un potencial atacante acceder a información sensible o confidencial que se encuentre almacenada en el portapapeles.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Como regla general se deberá deshabilitar el soporte a las funcionalidades de copy & paste de todos aquellos elementos de la interfaz susceptibles de representar información sensible o confidencial.

5.11 *Device Layer*. Uso de librerías de terceros.

Nativo	Híbrido	Web móvil
Crítico	Crítico	Crítico

El uso de librerías de terceros es un riesgo de seguridad debido a las potenciales vulnerabilidades o debilidades que podrían contener y que pondrían en riesgo la integridad de la aplicación y su información.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Será obligatorio el uso de las librerías homologadas en todas aquellas funcionalidades relativas a integración y persistencia.
- El uso de librerías de terceros que no se encuentren homologadas por DGT deberá ser previamente autorizado por el Departamento de Arquitectura, que en el caso de rechazar la autorización de uso deberá proporcionar la alternativa de uso.
- Todo proceso de homologación de una librería de terceros deberá contemplar una auditoría de seguridad como requisito imprescindible y cada actualización de una librería homologada (o versión de SO) debe contemplar la obligatoriedad de pasar un nuevo proceso de homologación como si de una nueva librería se tratara.

5.12 *Device Layer*. Uso de geolocalización

Nativo	Híbrido	Web móvil
Crítico	Crítico	Bajo

La posibilidad de que un potencial atacante se pueda hacer con la información del posicionamiento de un usuario representa un riesgo importante a la seguridad y privacidad.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices para aquellas aplicaciones que sean consideradas especialmente sensibles:

- Siempre que sea posible se deberá evitar guardar la información de posicionamiento (GPS) localmente en el dispositivo (*filesystem*, BBDD, logs…).
- Si ha de guardarse localmente información de posición (GPS) localmente se deberá realizar previa autorización del Departamento de Arquitectura y dicha información se deberá guardar siempre cifrada a través de criptografía fuerte y el uso de una *master key* que a su vez deberá estar protegida mediante una *password*, usando una *key derivation function* tal como se especifica en el punto 5.1.
- Siempre que sea posible se deberán eliminar los metadatos de posicionamiento de todos aquellos medios generados por el dispositivo (por ejemplo información EXIF en imágenes) antes de su persistencia local.

5.13 *Device Layer*. Permisos de aplicación.

Nativo	Híbrido	Web móvil
Crítico	Crítico	Bajo

En Android, el modelo de acceso de las aplicaciones a los recursos del dispositivo está basado en la aceptación, por parte del usuario, de los permisos parametrizados en el fichero de manifiesto de la aplicación. Esto hace que sea fácil para el usuario proporcionar su aceptación a permisos que la aplicación realmente no necesita y

que pueden representar un riesgo importante para la seguridad e integridad de la información de la aplicación.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- El uso de los permisos de MODE_WORLD_READABLE y/o MODE_WORLD_WRITABLE deberán ser autorizados previamente por el Departamento de Arquitectura ya que proporcionan amplios permisos de accesos al filesystem del sistema, incluido el directorio privado de datos de la aplicación.
- El manifiesto de la aplicación deberá incluir exclusivamente aquellos permisos que se necesiten para acceder a los recursos del dispositivo requeridos.
- Todos los permisos en desuso deberán ser eliminados de todas aquellas releases que puedan ser susceptibles de distribución fuera del ámbito del equipo técnico de desarrollo, sean estas releases de desarrollo o, con más importancia, de producción.

5.14 *Device Layer*. Implementación de Intents.

Nativo	Híbrido	Web móvil
Crítico	Crítico	Bajo

Los Intents en Android, tal como se especifica en el punto 4.9**,** son artefactos cuya responsabilidad es la comunicación entre componentes que forman parte de la arquitectura de Android y que pueden ser usados para:

- Iniciar una Activity, normalmente asociado a la apertura de una interfaz.
- Emitir un *broadcast* para informar al sistema operativo o a las aplicaciones de determinados cambios.
- Emitir eventos de inicio, finalización y comunicación con servicios.
- Acceso a información a través de *Content Providers*.
- Como un *callback* para la gestión de eventos (programación asíncrona).

Los componentes que pueden ser accedidos por *Intents* que pueden ser públicos (desde cualquier parte del sistema) o privados (exclusivamente desde la misma aplicación desde donde se emite el *Intent*).

Es un error común no restringir la configuración de acceso de los componentes accedidos a través de *Intents* de manera que dichos componentes acaban ofreciendo acceso fuera del ámbito de la aplicación lo que implica un importante riesgo de seguridad que podría ser aprovechado por un potencial atacante como vector para intentar acceder a información sensible o confidencial de la aplicación.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Como norma general se deberá procurar que todos los componentes de la aplicación solo puedan ser accedidos a través de *Intents* privados (solo emitidos desde dentro de la misma aplicación). Para ello se deberá utilizar la directiva:

android:exported="false"

- Si ha de publicarse un componente para que pueda ser accedido de manera pública se deberán definir permisos particulares para la aplicación, que deberán encontrarse declarados en el manifiesto de la aplicación.

- Como norma general toda la información recibida en un componente que pueda ser accedido de manera pública deberá ser considerada como información de riesgo y se deberá actuar en consecuencia, estableciendo todas aquellas validaciones que permitan asegurar la correcta integridad de los datos antes de ser usados.

- En aplicaciones Android, como norma general toda la información recibida a través de un *BroadcastReceiver* deberá ser considerada como información de riesgo y se deberá actuar en consecuencia, estableciendo todas aquellas validaciones que permitan asegurar la correcta integridad de los datos antes de ser usados. Se establecerán permisos para que sólo aplicaciones específicas puedan recibirlos.

- En aplicaciones Android, siempre que sea posible se deberá priorizar el uso de los artefactos de intercomunicación establecidos entre *Activities* y *Services* (*onBind*) sobre el uso de *Broadcas*t y *Broadcast Receivers*, al ser este método de comunicación público y accesible desde todo el sistema.

- Como norma general no se deberá incluir información sensible o confidencial dentro de *Intents* que se utilicen para inicializar *Activities*. Un potencial atacante podría insertar un *Intent Filter* con una mayor prioridad pudiendo recuperar la información.

5.15 Device Layer. Implementación de Activities.

Nativo	Híbrido	Web móvil
Crítico	Medio	Bajo

En aplicaciones Android, las *Activities* están íntimamente asociadas a las vistas, tal como se especifica en el punto 4.4. Las *Activities* pueden ser invocadas por cualquier aplicación si no se definen como privadas (sólo accesibles desde la aplicación). Esto, como en el caso del punto anterior, implica un importante riesgo de seguridad que podría ser aprovechado por un potencial atacante como vector para intentar acceder a información sensible o confidencial de la aplicación.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Como norma general todos los *Intent Filter* definidos en el manifiesto para una *Activity* deberán ser definidos explícitamente como privados.
- Como norma general se deberá utilizar los Intent en todas aquellas *Activities* que puedan ser accedidas desde fuera de la aplicación.
- Como norma general se deberá establecer una validación en todas las *Activities* que permita validar si la aplicación se encuentra en un estado compatible con un proceso válido de autenticación. El objetivo es prevenir que un potencial atacante pueda acceder a una *Activity* realizando un bypass del modelo de seguridad de la aplicación (esto puede realizarse, incluso con *Activities* definidas como privadas, si el modelo de seguridad del dispositivo ha sido comprometido).

- Como norma general toda la información recibida en una *Activity* que pueda ser accedida de manera pública deberá ser considerada como información de riesgo y se deberá actuar en consecuencia, estableciendo todas aquellas validaciones que permitan asegurar la correcta integridad de los datos antes de ser usados.

5.16 Device Layer. Implementación de Broadcasts.

Nativo	Híbrido	Web móvil
Crítico	Medio	Bajo

En aplicaciones Android, los *Broadcast* son un tipo de notificaciones que tienen la característica de propagarse por todo el sistema, tal como se especifica en los puntos 4.8 y 4.9, pudiendo ser atendidas por varios actores siempre que estos se encuentren registrados específicamente para la recepción de dichas notificaciones.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Como norma general se deberá evitar enviar información sensible o confidencial dentro de *Broadcasts*.

- Como norma general se establecerán permisos especiales asociados a *Broadcast* que deberán ser validados por el usuario y que dificultan la instalación de una aplicación que pueda registrar una escucha del *Broadcast* sin tener ningún tipo de dependencia o relación.

5.17 Device Layer. Implementación de PendingIntents.

Nativo	Híbrido	Web móvil
Medio	Medio	Bajo

En aplicaciones Android, los *PendingIntents* son artefactos que permiten a las aplicaciones pasar el flujo de ejecución a otra aplicación que podrá ejecutar el *Intent* registrado como si fuera la aplicación origen. Esto implica un serio riesgo de seguridad ya que la aplicación destino podría influir tanto en el destino como en la integridad de la información contenida en el *PendingIntent*.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Siempre que sea posible se deberá evitar el uso de *PendingIntents* en la aplicación.
- Si ha de publicarse un *PendingIntent* este deberá usarse principalmente como *callbacks* de *Broadcast Receivers/Activities* definidas como privadas y se deberá especificar en el *PendingIntent* de manera explícita el nombre del componente como uno de los componentes de la aplicación. Cualquier otro uso deberá ser autorizado previamente por el Departamento de Arquitectura.

5.18 *Device Layer*. Implementación de *Services*.

Nativo	Híbrido	Web móvil
Crítico	Medio	Bajo

En aplicaciones Android, los *Services* se utilizan habitualmente para la ejecución de procesos en *background* porque aunque se ejecutan en el *UI Thread* carecen de representación gráfica, tal como se especifica en el punto 4.6.

De manera similar a las *Activities* y a los *Broadcast Receivers*, los *Services* deben protegerse a través de la definición de permisos especiales y de restricciones de acceso (acceso privado) ya que comparten junto a estos los mismos riesgos de seguridad.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Todos los accesos a un *Service* que involucren información sensible o confidencial deberán establecer un proceso de validación que permita asegurar que el *Service* no ha sido suplantado (por ejemplo, especificando el nombre del componente que contiene la implementación del *Service* como un parámetro del *Intent* involucrado en la comunicación).

5.19 Device Layer. Implementación de Content Providers

Nativo	Híbrido	Web móvil
Medio	Medio	Bajo

En aplicaciones Android, los *Content Providers* se utilizan habitualmente como mecanismo para compartir información entre aplicaciones que, en un modelo de *sandbox* como el que implementa Android, se encuentran aisladas, tal y como se especifica en el punto 4.7.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Como norma general se deberá separar la asignación de permisos de lectura y escritura mediante la implementación de permisos diferenciados para cada operación.
- Como norma general solo se proporcionará el permiso de escritura cuando sea imprescindible.
- Como norma general se deberá realizar una implementación que aproveche la orientación del *Content Provider* al uso de registros para proporcionar la mínima información necesaria para cubrir el requerimiento funcional y así evitar exponer toda la información.

5.20 *Device Layer*. Implementación de *WebViews*.

Nativo	Híbrido	Web móvil
Medio	Crítico	Crítico

En la implementación de *WebViews* se deberán seguir las siguientes directrices:

- Como norma general se deberá deshabilitar de manera explícita el soporte de Javascript y de *plugins* si estos no son necesarios.
- Como norma general se deberá deshabilitar el acceso al filesystem local.
- Como norma general se deberá evitar la carga de contenido desde host que no se encuentren dentro de la DGT o que no hayan sido

autorizados previamente por el Departamento de Arquitectura. Esto se deberá realizar mediante la sobre escritura de los métodos “shouldOverrideUrlLoading” y “shouldInterceptRequest” que permitirá interceptar, inspeccionar y validar aquellas peticiones realizadas desde el *WebView*.

5.21 *Device Layer*. Caché de objetos gráficos.

Nativo	Híbrido	Web móvil
Medio	Bajo	Bajo

En aplicaciones Android las vistas por defecto se cachean en memoria para permitir un mejor rendimiento y experiencia de usuario cuando se solicita una vista que ha sido previamente accedida por el usuario. Esto puede representar un problema de seguridad porque un potencial atacante podría navegar por la aplicación y acceder a información sensible o confidencial de las diferentes vistas de la aplicación que puedan haber sido cacheadas.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Cuando el usuario realiza un cierre (“log out”) de la aplicación, se deberán de eliminar los datos sensibles que la aplicación tenga almacenado en memoria.

- Como norma general cada *Activity* deberá incorporar una implementación que permita comprobar si el estado de la aplicación es compatible con un proceso correcto de autenticación y de perdurabilidad de la sesión local. En caso contrario se deberá re

direccionar el flujo de navegación de la aplicación a la vista responsable de la autenticación de los usuarios.

5.22 *Device Layer*. Generación de UUID.

Nativo	Híbrido	Web móvil
Crítico	Crítico	Crítico

La generación y uso de un UUID implica un riesgo para la seguridad, privacidad e integridad de la información del usuario si el algoritmo utilizado no proporciona una disociación real entre el identificador y el dispositivo.

Por ello se deberá utilizar el algoritmo propuesto a continuación que asegura dicha disociación.

La base para ello es el uso de un número aleatorio que se deberá definir cuándo se inicia por primera vez el terminal y el número de serie que asigna el fabricante. Los datos numéricos de partida se someten a un proceso de degradado matemático (*hashing*) que reduce la resolución numérica, rompiendo de forma irreversible la relación del dato resultante con los datos originales.

Por lo tanto el primer dato (número aleatorio) se obtendrá a partir de:

public static final String ANDROID_ID

Que según la documentación de Google:

> *A 64-bit number (as hex string) that is randomly generated when the user first sets up the device and should remain constant for the lifetime of the user´s device. The value may change if a factory reset is performed on the device.*

> *Note: When a device has multiple users (available on certain devices running Android 4.2 or higher), each user appears as a completely separate device, so the ANDROID_ID value is unique to each user.*

El segundo dato (identificador del dispositivo) se obtiene a partir de la función *getDeviceId()* del SDK de Android y que según la documentación de Google:

> *Returns the unique device ID, for example, the IMEI for GSM and the MEID or ESN for CDMA pones. Return null if device ID is not available.*

Por último, como paso final para conseguir un resultado anónimo del conjunto se utilizan los *hash codes* de cada una de las partes para calcular el valor final que se calculará a partir de la siguiente expresión matemática:

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

Los caracteres s[0,n] se tratan como valores enteros.

La implementación en Java sería la siguiente:

```
/**
 *        Returns a hash code for this string. The hash code for a String object is computed as
 *                  s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
 *        using int arithmetic, where s[i] is the ith character of the string, n is the length of the
string, and ^ indicates exponentiation.
 * (The hash value of the empty string is zero.)
 *        Returns:
 *        a hash code value for this object.
 */
              public int function() {
                  int h = hash;
                  if (h == 0) {
                      int off = offset;
                      char val[] = value;
                      int len = count;

                      for (int i = 0; i < len; i++) {
                          h = 31*h + val[off++];
                      }
                      hash = h;
                  }
                  return h;
              }
Una implementación del algoritmo sugerido sería:
       public static String getDeviceId(Context context) {
              String id = getUniqueID(context);
              if (id == null)
                     id              =              Settings.Secure.getString(context.getContentResolver(),
Settings.Secure.ANDROID ID);
              return id;
       }

       private static String getUniqueID(Context context) {

              String telephonyDeviceId = "NoTelephonyId";
              String androidDeviceId = "NoAndroidId";

              // get telephony id
              try {
                     final           TelephonyManager           tm           =           (TelephonyManager)
context.getSystemService(Context.TELEPHONY SERVICE);
                     telephonyDeviceId = tm.getDeviceId();
                     if (telephonyDeviceId == null) {
                           telephonyDeviceId = "NoTelephonyId";
                     }
              } catch (Exception e) {
              }

              // get internal android device id
              try {
                     androidDeviceId                                                                    =
android.provider.Settings.Secure.getString(context.getContentResolver(),
                                  android.provider.Settings.Secure.ANDROID_ID);
                     if (androidDeviceId == null) {
                           androidDeviceId = "NoAndroidId";
                     }
              } catch (Exception e) {
              }

                     String id = getStringIntegerHexBlocks(androidDeviceId.hashCode()) + "-"
                                  + getStringIntegerHexBlocks(telephonyDeviceId.hashCode());

                     return id;
       }
```

5.23 *Network Layer*. Validación completa SSL/TLS.

Nativo	Híbrido	Web móvil
Crítico	Crítico	Crítico

La validación parcial o incompleta de los certificados SSL utilizados en el establecimiento de conexiones seguras es la principal causa de ataques de tipo *man in the middle*. Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- En ningún caso se podrá utilizar ninguna estrategia que implique el *bypass* del proceso de validación de certificados SSL que incorporan por defecto las plataformas (por ejemplo, En Android, mediante una sobre escritura vacía del *TrustManager*).

- Todos los certificados SSL utilizados durante el proyecto deberán ser válidos y estar correctamente firmados por una CA.

- Se deberán guardar las URLs dentro de la aplicación utilizando aquellas estrategias que dificulten su modificación. Por ejemplo, será preferible establecer las URLs dentro de una clase dentro del código fuente de la aplicación a realizarlo en un fichero de configuración (texto plano) como recurso *(asset)* de la misma.

- Se deberá usar *"Strict Transport Security HTTP header"* siempre que sea posible (desarrollo web móvil o híbrido con soporte para dicha directiva por parte del navegador). Esto evitará que el establecimiento de una conexión segura dependa exclusivamente del protocolo especificado en la URL (https/http) y que puede ser modificado con facilidad si la URL se encuentra en claro (ver punto anterior).

- No se deberá mostrar en la interfaz ningún tipo de símbolo o redacción externa a la implementada por defecto en el mecanismo de seguridad utilizado. De esa manera se evitará que el usuario tenga una errónea percepción de seguridad.

- En ningún caso se podrán utilizar certificados autofirmados.

- En ningún caso se podrán utilizar las directrices *org.apache.http.conn.ssl.AllowHostNameVerifier* o *SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER.*

5.24 *Network Layer*. Comunicaciones seguras.

Nativo	Híbrido	Web móvil
Crítico	Crítico	Crítico

En el desarrollo de aplicaciones móviles la mayor parte de integraciones se realizan a través de protocolos como HTTP o FTP. Estos protocolos, por defecto, no incorporan sistemas de seguridad que permitan asegurar la confidencialidad e integridad de la información transmitida y esto representa un riesgo alto de seguridad por el cual un atacante podría modificar, degradar o duplicar la información transmitida.

Teniendo esto en cuenta, en este punto se deberán seguir en todo momento las siguientes directrices:

- Como norma todas las integraciones deberán encontrarse protegidas mediante el uso de las versiones seguras de los protocolos

involucrados (HTTP(s), FTP(s)…) independientemente del nivel de sensibilidad de la información intercambiada.

- Será responsabilidad del Departamento de Arquitectura definir qué información requiere de una capa de seguridad extra (por ejemplo mediante encriptación simétrica del mensaje) independiente de la protección establecida en el canal de transmisión.

- Si ha de realizarse una integración a través de un protocolo que no incorpore ningún sistema de seguridad de encriptación del canal, esta deberá ser aprobada previamente por el Departamento de Arquitectura, independientemente de que la integración se realice con servicios internos de DGT o de terceros.

5.25 *Network Layer*. Gestión de sesiones y SSO.

Nativo	Híbrido	Web móvil
Crítico	Crítico	Crítico

Debido a la naturaleza *stateless* de los protocolos de integración utilizados habitualmente (HTTP(s) para servicios REST o SOAP) es importante habilitar en los servicios publicados en el *backend* un sistema de sesión que permita identificar de manera unívoca a un usuario previamente autenticado en la plataforma. Permitiendo de esa forma la escalabilidad en el acceso a la información y en las operaciones disponibles a través de los servicios publicados.

El conjunto de aplicaciones que necesiten de autenticación por parte del usuario para el uso de la misma conformarán un dominio de Single Sign-On. El acceso a la primera de ellas cuando no haya una sesión establecida desencadenará el proceso

de login y la creación de la sesión. Será necesario que en el dispositivo se encuentre instalada la aplicación agente de SSO y que las aplicaciones usen el SDK de autenticación que interactúa con la misma. Las aplicaciones simplemente solicitarán una sesión usando el SDK y obtendrán un objeto de sesión con la información correspondiente al usuario y un token que deberán adjuntar en cada petición a servicios de backend que realicen. Dicho token es temporal, siendo necesario solicitar uno nuevo cuando caduque. El modelo de autenticación está basado en OAuth. Los detalles se ofrecen en el punto 6.

Aquellas aplicaciones que no formen parte de un dominio SSO y necesiten invocar a recursos protegidos de backend deberán obtener las credenciales de invocación a los mismos en función del tipo de protección que dichos servicios de backend tengan establecidos en cada caso.

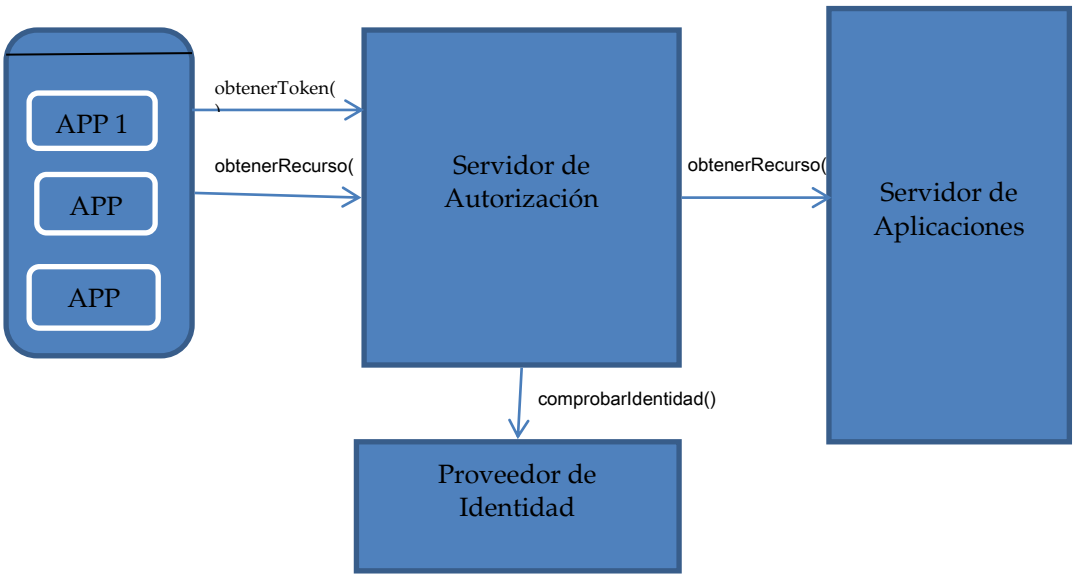
6 Single Sign-On y protección de recursos backend

Tanto el entorno de SSO como la protección de recursos de backend en el ESB estarán basados en OAuth 2. En concreto, los token de refresco de OAuth son fundamentales a la hora de manejar situaciones de falta de cobertura que obliguen a posponer el envío de peticiones de recursos al backend. Los recursos de backend estarán protegidos con un modelo *n factor authentication*, en línea con el tipo de clientes que los usarán; en concreto, aparte de la identificación del usuario (vía token OAuth o cualquier otro tipo de credencial que justificadamente pueda ser necesaria en ciertos recursos de backend) será necesario identificar a la aplicación y al dispositivo, de forma que pueda asegurarse que la invocación al servicio de backend procede de una aplicación autorizada que está instalada en un dispositivo previamente registrado y la está usando un usuario autorizado previamente autenticado. En ciertos escenarios puede ser incluso necesario introducir elementos adicionales basados en el contexto (como por ejemplo la ubicación física o el instante de tiempo).

En los dispositivos móviles estarán disponibles aplicaciones desarrolladas tanto por la DGT como por otros organismos, siendo necesario que todas ellas formen parte del mismo contexto SSO y que las aplicaciones desarrolladas por terceros funcionen también en los dispositivos de sus empleados. El protocolo OAuth necesita de un servidor de autorización que emita los *token* y que compruebe la validez de los mismos cuando se acompañan a las peticiones de recursos protegidos, el cual se apoyará en un proveedor de identidad para comprobar las credenciales de los usuarios; el servidor de autorización también tendrá que comprobar aquellos

elementos credenciales dentro del modelo *n factor authentication* que se necesita seguir, para lo cual podrá apoyarse tanto en el proveedor de identidad como en cualquier otro elemento necesario para la comprobación de dichos elementos credenciales. Y dado que las aplicaciones necesitarán recursos del *backend*, los servidores de aplicaciones también forman parte del escenario. Resumiendo, los elementos involucrados serían los siguientes:

- Las aplicaciones que formen parte del mismo contexto SSO (con independencia de su procedencia)
- El servidor de autorización
- El/los proveedor/es de identidad
- Los servidores de aplicaciones



Este escenario obliga a que el diseño de las aplicaciones esté orientado a una independencia con respecto al contexto en el que tengan que ejecutarse, evitando dependencias y acoplamiento con respecto a los tipos de *token* a manejar y con respecto al servicio de autorización que emita dichos *token,* e incluso con respecto a

los *endpoint* de los recursos que necesiten ser invocados. Por otra parte será necesario disponer de un mecanismo que presente el formulario de *login* en sustitución de un navegador (tanto externo como embebido en la aplicación). Se opta en consecuencia por definir un SDK que usarán las aplicaciones para obtener tanto los *token* de acceso a los recursos como para obtener la información del usuario; la aplicación sólo necesitará conocer dicho SDK, el cual será una abstracción que desacople a las aplicaciones del contexto en que se ejecutan. El SDK interactuará con una aplicación nativa independiente que se encargará de presentar el formulario de *login* y que será quien interactúe con el servidor de autorización de forma que el *token* y el propio servidor de autorización no tengan necesidad de ser conocidos por las aplicaciones; dicha aplicación nativa se encargará de servir al SDK las implementaciones de las abstracciones que representan a los protocolos concretos del servidor de autorización, las cuales permitirán construir las peticiones de recursos protegidos hacia el *gateway*. Esta abstracción permitirá cambios transparentes para las aplicaciones en la infraestructura de SSO y que las aplicaciones puedan funcionar en diferentes contextos en los que los servidores de autorización y los *token* (en definitiva, la infraestructura SSO) sean completamente distintos, permitiendo que cada organismo que desarrolle aplicaciones que quiera exportar para su uso por terceros no se vea condicionado por los contextos y la infraestructura de organismos ajenos. Tan sólo hay que estandarizar un SDK, el cual no dependerá de ningún agente concreto sino que simplemente esperará disponer de una implementación del agente que cumpla con las interfaces definidas en el SDK. Compartiendo el mismo SDK, cada organismo podrá desarrollar su agente de forma independiente (o incluso implementar enfoques distintos). Si las aplicaciones de terceros, al funcionar en contextos distintos al original, necesitan obtener recursos de su propio *backend* es posible que sea necesario realizar una conversión de *token* previa, lo cual puede hacerse disponiendo de un *gateway* que se encargue de la conversión, siguiendo un

patrón tipo *Adapter* o un modelo de conversión de protocolos en función de los requisitos de seguridad de los recursos a solicitar. En el caso de los *backend* propios de la Dirección General de Tráfico, siempre será necesario generar un nuevo *token* partiendo de la información obtenida del *token* OAuth. Esta indeterminación con respecto a los *endpoint* obliga a resolver en tiempo de ejecución los mismos, para lo cual será necesario establecer un mecanismo de resolución que consistirá en la existencia de un registro del que puedan obtenerse los *endpoint* de los recursos; dicho registro tendría que existir en cada contexto.

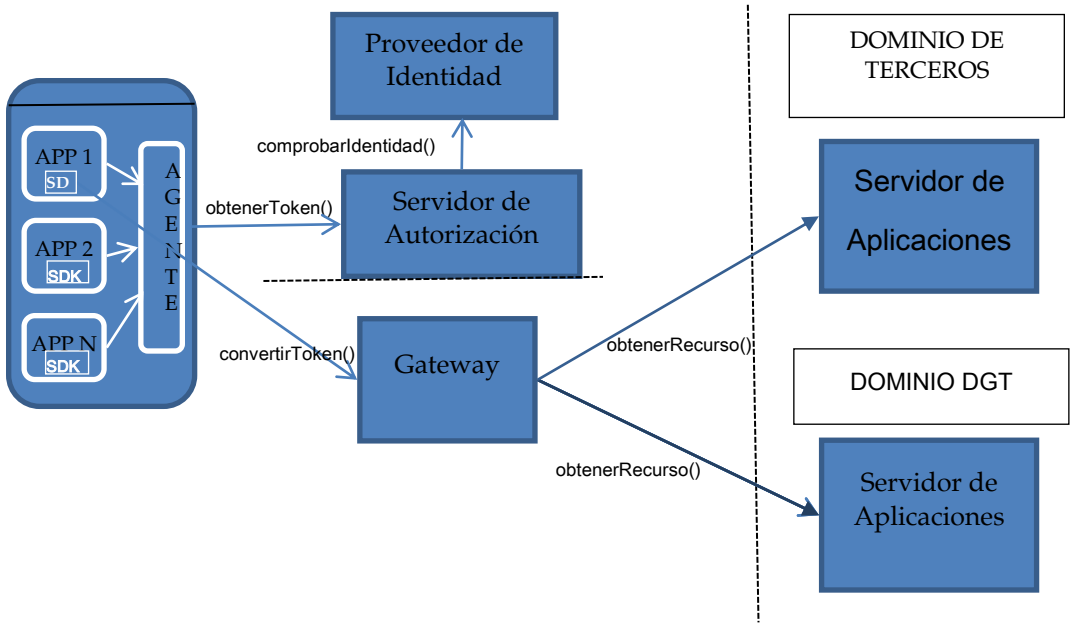
Con estos nuevos elementos la relación de elementos involucrados quedaría como sigue:

- Las aplicaciones que formen parte del mismo contexto SSO (con independencia de su procedencia)
- El servidor de autorización
- El/los proveedor/es de identidad
- Los servidores de aplicaciones
- Una aplicación agente encargada de la interacción con el servidor de autorización, de la presentación del formulario de *login,* proporcionar las implementaciones de protocolo concretas, obtener las aplicaciones autorizadas para cada usuario y los *endpoint* de los recursos que necesita cada aplicación
- Un SDK que abstraiga a las aplicaciones del contexto SSO (agente, *token*, servidor de autorización)
- Un *gateway* que transforme las peticiones a *backend* para que vayan con el tipo de *token* esperado por cada *backend*

Nos estamos refiriendo tanto al servidor de autorización como al *gateway* como elementos lógicos dentro del escenario, pero físicamente puede tratarse de un mismo elemento que represente ambos roles, tal y como se detallará más adelante en el contexto de la Dirección General de Tráfico.

La solución basada en SDK y aplicación nativa agente es una solución que se está adoptando de forma masiva en los entornos de movilidad y es la solución usada

para evitar los flujos web en las invocaciones a recursos protegidos desde aplicaciones móviles nativas en muchas compañías que ofrecen servicios a través de API públicas (Google, Facebook, LinkedIn y muchas otras). También se utiliza el mismo enfoque para soluciones de SSO como representante del servidor en el dispositivo, ofreciendo de forma local servicios que de otro modo habría que invocar con flujos web con la consiguiente merma en la usabilidad. Asimismo, es el enfoque en el que se está basando un nuevo estándar de OpenID para establecer mecanismos de SSO en aplicaciones nativas para móviles (*OpenID Connect Native Application Token Agent*).



6.1 Contexto de la Dirección General de Tráfico

A continuación se pasa a detallar los elementos concretos que forman el contexto de la Dirección General de Tráfico.

Tanto el SDK como el agente son elementos a desplegar en cada dispositivo. El agente como aplicación independiente y el SDK como librería embebida en cada aplicación.

El servidor de autorización, el *gateway* y el proveedor de identidad serán funciones ejercidas por el dispositivo IBM Datapower XI52 (firmware 7.0) y un directorio LDAP, en concreto Sun One Directory Service. Como servidor de autorización ejercerá las dos funciones definidas en el protocolo OAuth: servidor de autorización (generará los *token* una vez comprobada la identidad del usuario) y punto de comprobación de acceso a recursos (comprobará la validez de los *token* aportados al solicitar recursos). Como *gateway* se encargará de generar los *token* que los servicios de *backend* necesiten transformando el *token* OAuth recibido en el *token* esperado por cada servicio de *backend*. Como proveedor de identidad comprobará tanto las credenciales de los usuarios como el resto de parámetros aportados dentro de un modelo *n factor authentication*. En el escenario de integración con organismos externos, será necesario que usuarios ajenos a la Dirección General de Tráfico (que no figuran en su registro de usuarios) sean capaces de autenticarse en las aplicaciones, para lo cual pueden plantearse dos posibilidades:

1. Contar con un solo proveedor de identidad. Todas las aplicaciones usarán el proveedor de identidad del contexto en el que se ejecuten, para lo cual el proveedor de identidad y/o alguno de sus elementos auxiliares tendrá que estar federado con el registro de usuarios ajeno.
2. Contar con más de un proveedor de identidad. El servidor de autorización deberá en este caso determinar en qué proveedor de identidad debe comprobar las credenciales, usando estrategias *round robin* o cualquier otras.

No es objeto de este documento establecer qué posibilidad a usar, ya que es algo que no afecta a la solución de SSO, pudiendo adoptarse cualquier estrategia a tal efecto.

El registro de *endpoint* podrá implementarse directamente en el ESB o en el servidor de aplicaciones.

6.2 Registro de factores de autenticación adicionales (*n factor authentication*)

6.2.1 Registro de dispositivos

Para que sea posible un modelo de *n factor authentication* es necesario tener identificados y registrados los factores adicionales a los típicos valores de nombre de usuario y *password*, los cuales se encuentran ya registrados en el repositorio de usuarios. Uno de los factores adicionales a tener en cuenta será el identificador de dispositivo, que servirá para asegurar que las aplicaciones autorizadas se están ejecutando desde un dispositivo autorizado. Como identificador de dispositivo se descarta el IMEI y cualquier otro identificador que no sea secreto. Lo que se hará es combinar varios de esos valores de forma que sea difícil deducir qué combinación de valores es la necesaria. Asimismo, es necesario que exista una disociación real entre el identificador generado y el dispositivo. La base para ello es el uso de un número aleatorio que se deberá definir cuándo se inicia por primera vez el terminal y el número de serie que asigna el fabricante. Los datos numéricos de partida se someten a un proceso de degradado matemático (*hashing*) que reduce la resolución numérica, rompiendo de forma irreversible la relación del dato resultante con los datos originales.

Se usará una combinación del ANDROID_ID y el IMEI. ANDROID_ID es un valor de 64 bits que se genera aleatoriamente al inicializar el dispositivo. Dado que el ANDROID_ID no puede ser obtenido consultando la configuración del dispositivo, y también debido a la dificultad de gestionar el registro de miles de dispositivos, se necesitará un mecanismo de registro dinámico, siendo el agente el encargado de registrar el dispositivo si aún no se ha realizado el registro. Para ello es necesario

que el servidor de autorización ofrezca un servicio de registro con funciones de consulta de registro y de registro. Para asegurar que el registro no lo realiza un agente no autorizado el agente tendrá que identificarse debidamente como aplicación registrada, para lo cual se aportará un certificado de cliente al *endpoint* de registro de dispositivos.

El registro de dispositivos lo realizará la aplicación agente, no siendo necesario que el resto de aplicaciones hagan nada al respecto.

6.2.2 Registro de aplicaciones

Para asegurar que las aplicaciones que solicitan *token* son aplicaciones autorizadas, aparte del mecanismo establecido por el propio protocolo OAuth para la identificación de aplicaciones se usarán factores adicionales de identificación de las mismas. Asimismo estos factores adicionales servirán para que las aplicaciones y el agente confíen mutuamente entre sí y el flujo de los *token* esté asegurado contra elementos no autorizados.

Los factores adicionales serán el *hash key* con que se firman las aplicaciones y el nombre del paquete base de las mismas.

Aparte de los factores adicionales de autenticación se registrará de cada aplicación la relación de los *endpoint* de los servicios de negocio a los que tendrá que acceder, de cara a parametrizar los mismos con objeto de que las aplicaciones puedan adaptarse al entorno en el que se ejecutan sin suponer que los *endpoint* serán los mismos en todos los entornos, así como la relación de usuarios autorizados a usar la misma. Asimismo será necesario registrar la relación de roles autorizados a usar la aplicación.

Este registro servirá no sólo para reforzar el proceso de autenticación, sino también para facilitar la obtención de los *token* de aplicación al inicio de sesión.

6.3 Inicio de aplicaciones

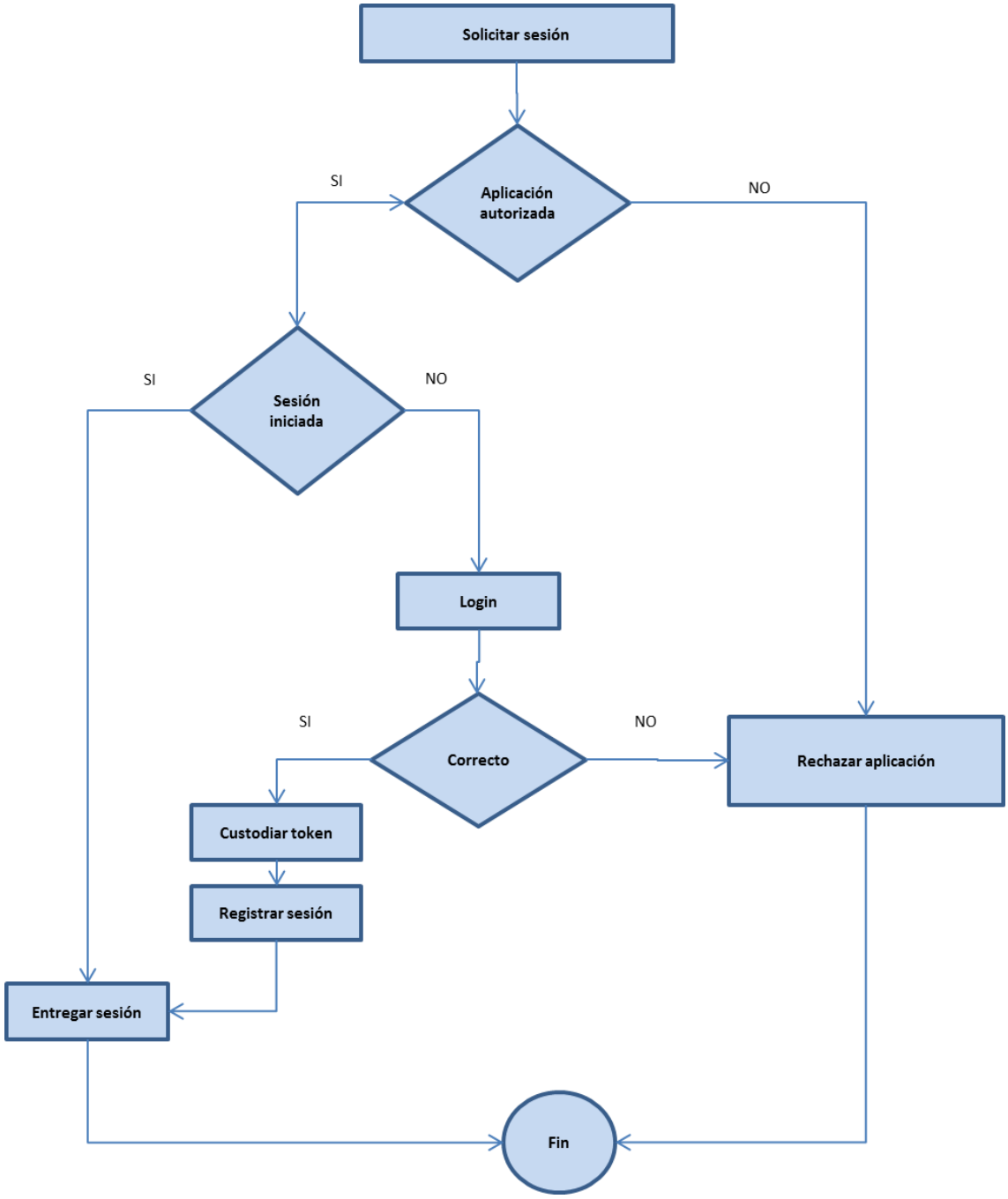
A continuación se pasa a detallar el flujo que deben seguir las aplicaciones incluidas en el dominio SSO cuando se inician (queda incluida como aplicación dentro de dicho dominio el agente, ya que se trata de una aplicación más a partir de cuyo inicio puede desencadenarse un inicio de sesión):

1. El usuario inicia su interacción con una aplicación incluida en el dominio SSO.
2. La aplicación, a través del SDK, solicitará al agente un objeto que represente la sesión (y que contendrá asimismo un objeto que represente la identidad del usuario, así como la relación de *endpoint* de cada aplicación y cualquier otra información necesaria). Deberá aportar un *callback* para ser informada del resultado del proceso e identificarse ante el agente con su identificador (*scope*).
3. El agente, si tiene una sesión ya iniciada y no caducada, devolverá el objeto asociado a dicha sesión y la aplicación dispondrá de la información sobre el usuario propietario de la sesión, pudiendo éste usar la aplicación a partir de ahí. Previamente el agente habrá comprobado que la petición proviene de una aplicación autorizada remitiendo el *hash key* al servidor de autorización para su comprobación (una validación incorrecta provocará el fracaso del inicio de sesión). Se comprobará que el usuario tenga permiso para usar la aplicación (usando el rol del mismo), y en caso contrario no se proporcionará el objeto de sesión y la aplicación en consecuencia no permitirá al usuario el uso de la misma.
4. Si el agente no tiene una sesión válida presentará al usuario un formulario de *login* y, tras introducir el usuario las credenciales, invocará al *endpoint* de inicio de sesión del servidor de autorización con dichas credenciales, su *client ID*, su *secret,* el identificador del dispositivo, su *hash key* y su nombre de paquete para que valide todos esos elementos y obtener un *token* primario y un *token* de refresco asimismo primario en caso de ser todos correctos. Si algún elemento no es correcto (o si el usuario no está autorizado a usar la aplicación) no podrá iniciarse sesión y en consecuencia no podrá usarse la aplicación. Ambos *token* primarios obtenidos serán de uso exclusivo por el agente. Una vez obtenidos los *token* el agente debe descartar las credenciales del usuario.
5. El agente, una vez en su poder el *token* primario y el de refresco, invocará al *endpoint* de consulta de aplicaciones con su *token* primario, el nombre de pa-

quete, la *hash key* y el identificador de dispositivo para obtener la información de todas las aplicaciones que el usuario tiene derecho a usar. Si es necesaria la integración de aplicaciones de terceros en la respuesta se incluirán los *endpoint* de cada servicio de negocio que necesite invocar cada aplicación, de forma que el agente pueda proporcionar a las aplicaciones los *endpoint* adecuados al entorno en que se ejecutan. Se indicará como parámetro el esquema de respuesta deseado; en principio existirá sólo uno (*napps*) que opcionalmente incluirá los *endpoint* de cada servicio de negocio que necesite invocar cada aplicación.

6. Una vez obtenida la información de las aplicaciones el agente solicitará, usando su *token* de refresco, el *token* secundario para cada una de las aplicaciones indicando en una cabecera llamada *Application* el identificador de la aplicación de forma que el servidor de autorización pueda determinar el *scope* del *token* emitido (el identificador de la aplicación formará parte de la información de cada aplicación devuelta por el *endpoint* de consulta de aplicaciones). El agente se autenticará ante el servidor de autorización aportando, aparte de sus credenciales, todos los n factores de autenticación definidos (nombre de paquete, *hash key* e identificador de dispositivo). El agente deberá ofrecer un servicio a las aplicaciones para que éstas puedan obtener la relación de *endpoint* que necesitan o establecer un mecanismo similar para que las aplicaciones puedan obtener dicha relación (por ejemplo el agente podría registrarse como manejador de las URL que cumplan con un esquema determinado). Junto con el *token* secundario para la aplicación se recibirá un nuevo *token* de refresco, debiendo descartarse el anterior (que al usarse queda invalidado); es decir, con cada uso del *token* de refresco se redcibe un nuevo *token* de refresco que pasa a ser el actual y que sólo podrá ser usado una vez.
7. El agente custodiará todos los *token* de forma segura (y no volátil para garantizar pérdidas al apagar el dispositivo) y construirá el objeto de sesión, que proporcionará a las aplicaciones y que servirá como indicador de que existe una sesión iniciada. En la sesión figurará la identidad del usuario, su perfil y cualquier otro dato que se considere necesario. Asimismo incorporará la sesión al registro de sesiones, permaneciendo allí hasta que la sesión caduque y no queden operaciones pendientes pertenecientes a dicha sesión. La implementación concreta del registro de sesiones (SQLite, Shared Preferences…) vendrá determinada por la criticidad de la información que se guarde en la sesión.

8. El SDK invocará la función *callback* para informar a la aplicación del resultado del proceso.
9. En ambos casos el agente tendrá que devolver el flujo hacia la aplicación. Para prevenir intentos de *hijacking*, el SDK, antes de enviar el *Intent*, comprobará que el *Intent* se dirige al agente apropiado comprobando la *hash key* del agente.



Corresponderá al agente el control de caducidad de la sesión en base a la duración establecida, aparte de que el servidor pueda proporcionar una respuesta de error

cuando se le remita un *token* caducado. A tal efecto en la respuesta del servidor, aparte del *token* se incluirá la fecha de expiración del mismo (tal y como marca el protocolo OAuth) la cual servirá como fecha de expiración de la sesión, quedando de este modo configurada en el servidor la duración de la sesión, cuyo cambio no tendría impacto en el agente. El control de caducidad se establecerá en las peticiones que se hagan al agente de *token* para uso por las aplicaciones, ya que un control basado en cálculos de tiempo puede resultar molesto para el usuario al saltar un formulario de *login* de forma inesperada, así como complicar las operaciones de invocación al *backend*.

6.4 Invocaciones al *backend*

Cuando las aplicaciones necesiten invocar a los recursos protegidos del *backend* se seguirá el siguiente flujo:

1. La aplicación, a través del SDK, solicitará al agente un *token* para invocar al *backend*, aportando su identificador de aplicación.
2. El agente, que habrá obtenido el *token* durante el inicio de sesión, devolverá el *token* a la aplicación solicitante. Previamente habrá incrementado el contador de entregas correspondientes a dicho *token* en esa sesión.
3. La aplicación usará el *token* recibido para invocar al *backend* solicitando un recurso, y dicho *token* servirá para acceder a cualquier recurso que dicha aplicación necesite, ya que se habrá incluido en el *token* el conjunto de ámbitos necesarios. El agente proporcionará el mismo *token* para la misma aplicación mientras dure la sesión y el *token* no esté caducado. El cliente habrá de autenticarse como tal ante el proveedor del recurso (siguiendo el modelo *n factor authentication*), al igual que lo hace el agente en cualquier petición al servidor de autorización, pero no siendo necesario en este caso aportar credenciales de cliente. El agente proporcionará a las aplicaciones una implementación de protocolo que sabrá cómo solicitar el recurso al servidor concreto y cómo interactuar con el agente para mantener la sincronización entre los *token* emitidos y los *token* usados. Al remitir la petición del recurso se comunicará al agente el éxito de la petición, en su caso, junto al *token* para que pueda disminuir el contador de dicho *token*. Para asegurar el adecuado con-

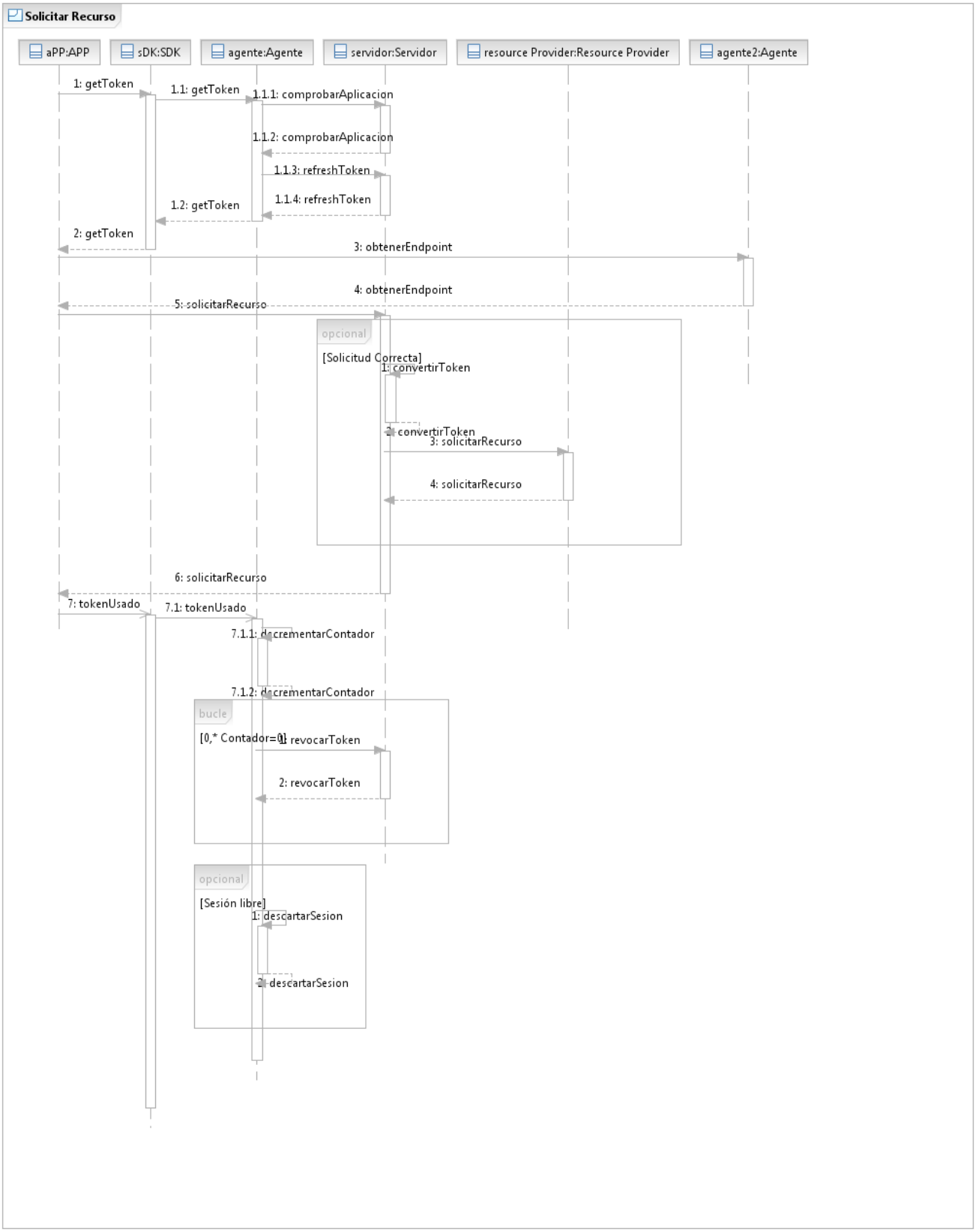
trol de las operaciones pendientes de realizar con dicho *token*, el *token* no debe ser guardado por la aplicación cliente, siendo necesario en este caso solicitarlo al agente en cada ocasión en que se necesite (lo cual asegurará que se cuenta con un *token* no caducado).

4. El proveedor del recurso estará protegido detrás del servidor de autorización, que validará el *token* y proporcionará el recurso. De no ser válido el *token*, se enviará una respuesta de error y la aplicación tendrá que pedir al agente un nuevo *token*, que será proporcionado usando el *token* de refresco de la sesión actual sin incrementar el contador.

```
HTTP/1.1 400 Bad Request
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
        "error":"error code"
        "error_description":"invalid_request"

}
```

5. Si no hubiese conectividad, o se decidiese enviar la petición a una cola, se guardará la petición junto con el *token* para su remisión posterior. Al remitir la petición del recurso se comunicará al agente el éxito de la petición, en su caso, junto al *token* para que pueda disminuir el contador de dicho *token*; si el *token* estuviese caducado (lo cual puede observarlo la propia aplicación o puede saberse por respuesta del servidor) se pedirá uno nuevo al agente indicando el *token* expirado (el agente lo proporcionará usando el *token* de refresco de la sesión correspondiente al *token* caducado sin incrementar el contador). El agente deberá comprobar si la aplicación que le solicita un nuevo *token* es la misma que recibió el anterior.


Solicitar Recurso
aPP:APP
sDK:SDK
agente:Agente
servidor:Servidor
resource Provider:Resource Provider
agente2:Agente
1: getToken
1.1: getToken
1.1.1: comprobarAplicacion
1.1.2: comprobarAplicacion
1.1.3: refreshToken
1.1.4: refreshToken
1.2: getToken
2: getToken
3: obtenerEndpoint
4: obtenerEndpoint
5: solicitarRecurso
opcional
[Solicitud Correcta]
1: convertirToken
2: convertirToken
3: solicitarRecurso
4: solicitarRecurso
6: solicitarRecurso
7: tokenUsado
7.1: tokenUsado
7.1.1: decrementarContador
7.1.2: decrementarContador
bucle
[0,* Contador=0]
1: revocarToken
2: revocarToken
opcional
[Sesión libre]
1: descartarSesion
2: descartarSesion

6.5 Caducidad de la sesión

El inicio de la sesión viene marcado por la obtención del *token* primario gestionado y custodiado por el agente. Es la fecha de expiración de dicho *token* la que marca la caducidad de la sesión. Como ha quedado dicho anteriormente, la sesión se descartará cuando el agente, al tener que hacer uso de dicho *token* para comunicarse con el servidor de autorización por cualquier causa, pueda comprobar que el *token* está caducado, o bien cuando el servidor de autorización le responda con un error por caducidad de *token*. Tanto si se hace de una forma o de otra, el agente descartará la sesión (marcándola como expirada), solicitará al servidor de autorización la revocación de los *token* primario y de refresco (en ese mismo momento o de forma diferida, dependiendo de si quedan peticiones al *backend* pendientes asociadas a la sesión), presentará el formulario de *login* e iniciará una nueva sesión obteniendo nuevos *token* primario y de refresco, así como el *token* secundario para la aplicación en su caso.

Como queda dicho, deberán seguir custodiándose sesiones antiguas mientras existan operaciones pendientes asociadas a ellas. A medida que el SDK, a través de la implementación de protocolo suministrada por el agente, vaya remitiendo comunicaciones de éxito de petición de recursos y el agente vaya disminuyendo los contadores correspondientes podrán comprobarse todos los contadores de la sesión y, en caso de estar todos a cero, descartar definitivamente la sesión y solicitar la revocación de todos sus *token* asociados (o bien el agente, al cerrar sesión, obtenga de las aplicaciones la confirmación del descarte de los *token*).

6.6 Logout

Dado que en algunos escenarios el dispositivo puede ser compartido por más de un usuario, será necesario en dichos escenarios que cuando un usuario deje de usarlo

cierre explícitamente la sesión de forma que se destruyan todos los *token* asociados a su identidad, así como el objeto de sesión.

A tal efecto, el agente debe ofrecer una función de cierre de sesión, en cuya implementación realizará la misma operativa que en la caducidad de sesión. De este modo, al no ser necesario distinguir si la nueva sesión corresponde a un nuevo usuario o al mismo, se simplifica todo el proceso.

6.7 Situaciones de falta de cobertura

6.7.1 Invocaciones al *backend* en segundo plano

Los dispositivos móviles se conectan a los servidores de *backend* a través de redes inalámbricas, cuya cobertura depende de la ubicación física concreta del dispositivo en cada instante, la cual cambia constantemente. Puede darse el caso en que, en un momento dado, se intente conectar con un servicio *backend* y no sea posible por falta de cobertura. En esos casos será necesario almacenar la petición para poder ser remitida posteriormente, cuando se recupere la conectividad (o bien puede establecerse como estrategia que todas las invocaciones al *backend* se almacenen para su remisión en segundo plano); como la recuperación de la conectividad no puede garantizarse a corto plazo es posible que cuando se recupere se haya iniciado una sesión con un usuario distinto, lo cual constituye un problema porque es necesario que las invocaciones al *backend* queden registradas en el mismo con la identidad del usuario que generó la petición en su momento, y no con la identidad del usuario en cuya sesión se consigue conectar con el *backend*.

Para solventar esta cuestión habrá de utilizarse la operativa indicada en el punto 5 del apartado 6.4.

6.7.2 Logout

La eliminación de la sesión local puede hacerse sin conectividad, pero la revocación de los *token* en el servidor de autorización tendrá que almacenarse para su remisión posterior en segundo plano.

6.8 Integración de aplicaciones de terceros

Puede ser necesario que aplicaciones desarrolladas por otros organismos para su propio entorno funcionen en los dispositivos de la Dirección General de Tráfico y se integren en el entorno de SSO con el resto de aplicaciones desarrolladas por la Dirección General de Tráfico.

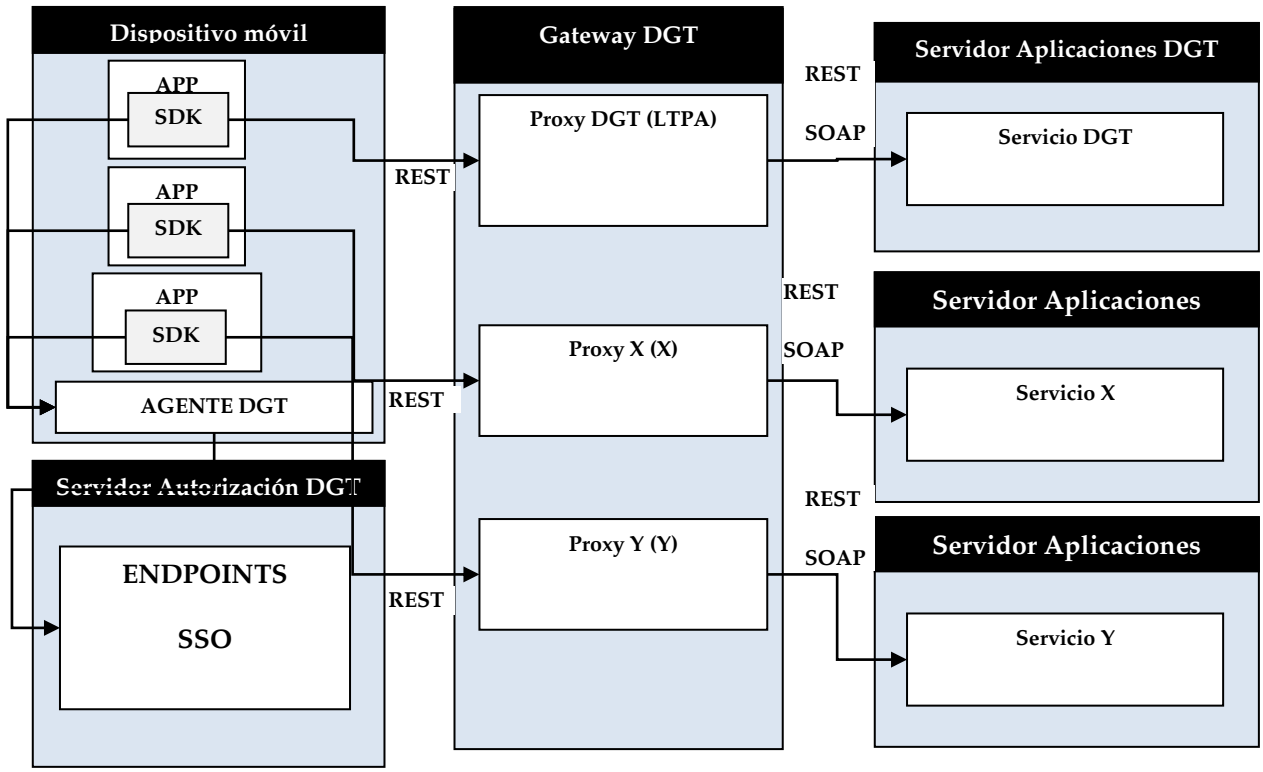
El propósito de esta recomendación en este escenario es evitar el acoplamiento y no condicionar el desarrollo de las aplicaciones para cada entorno. Cada organismo debe poder decidir qué tipo de aplicaciones usa, cómo las quiere construir y qué tipo de infraestructura de servidores y soporte SSO quiere usar. Sin embargo, algo en común han de tener para poder adaptarse al entorno en el que se ejecutan. Esa capacidad de adaptación es la base sobre la que se sustenta la solución presentada en este documento. Con la debida abstracción, las aplicaciones han de poder adaptarse al entorno y recibir de él los elementos necesarios para su adaptación. Lo que se propone es compartir el SDK de SSO, que se construirá de forma que sea agnóstico con respecto a la infraestructura de SSO existente en el entorno y con respecto a los *token* usados. Las concreciones estarán en el agente, que servirá de elemento integrador (en cada entorno deberá existir un agente que cumpla con el SDK definido). De este modo, las aplicaciones desarrolladas por cada organismo, si se apoyan en el mismo SDK, podrán funcionar en otros contextos ya que los agentes también se construirán basados en el mismo SDK, proporcionando a éste los elementos concretos que, implementando las interfaces definidas en el SDK, ofrecerán las implementaciones de protocolo concretas de cada entorno.

La clave pues se encuentra en la abstracción que ofrezca el SDK, que repetimos ha de ser agnóstico con respecto a los *token* y a la infraestructura de SSO. El agente se encargará de interactuar con el servidor de autorización y de obtener todos los *token*, siendo éstos para el SDK simplemente un elemento que debe adjuntar en las peticiones al *backend*. Sin embargo, dado que las peticiones al *backend* las realizarán directamente las aplicaciones (o a través de un elemento concentrador, que será ajeno al agente de SSO), es necesario que se proporcione a las mismas (o al SDK como elemento integrante de ellas) una implementación de protocolo de forma que las aplicaciones no tengan que conocer nada concreto del contexto y con ello se mantengan independientes de él y no acopladas al mismo; dicha implementación tendrá que ser suministrada por el agente.

Asimismo es necesario abstraer los *endpoint* de los recursos que necesiten invocar las aplicaciones. Al funcionar en contextos diferentes dichos *endpoint* pueden ser distintos ya que el camino a seguir para la obtención de los recursos puede necesitar la disposición de elementos intermedios (*gateways*) para realizar conversión de *token*, resolver problemas de firewall o por cualquier otra circunstancia. Para facilitar esa abstracción en los inicios de sesión el servidor de autorización facilitará, como parte del inicio de sesión, los *endpoint* de cada recurso necesario. El *endpoint* del registro se encontrará protegido del mismo modo que el *endpoint* del registro de dispositivos.

De este modo, al apoyarse en abstracciones, las aplicaciones podrán funcionar con independencia del contexto, que será el encargado de entenderse con la infraestructura concreta. Es un concepto similar a la arquitectura *Java Enterprise Edition*, en la que las aplicaciones implementan los requisitos propios del negocio y delegan en la infraestructura todas las cuestiones ajenas al mismo, adaptándose a los servicios ofrecidos por la misma y siendo agnósticas con respecto a cómo se implementan dichos servicios. Basando su implementación en las interfaces

definidas en un API consiguen integrarse en distintos servidores los cuales implementan libremente la parte del API correspondiente al servidor.



6.9 Consideraciones de seguridad

Hay que tener en cuenta ciertas consideraciones para garantizar la confidencialidad y mitigar los riesgos que tiene la obtención remota de *token* de seguridad, así como su custodia, máxime teniendo en cuenta la naturaleza pública tanto de los dispositivos como de las aplicaciones instaladas en ellos.

6.9.1 Confidencialidad en las comunicaciones

Todas las comunicaciones, ya sean con el servidor de autorización (en todos sus *endpoint*), con el *gateway* o con los servidores de aplicaciones se realizará con

seguridad a nivel de capa de transporte, en concreto usando TLS. Esto es normativo para todas las aplicaciones y para los proxys a desplegar en el *gateway*.

Aparte, las aplicaciones desplegadas en los dispositivos móviles habrán de comprobar siempre el certificado del servidor para asegurar que la respuesta proviene de un servidor de confianza.

6.9.2 Asegurar la comunicación entre las aplicaciones y el agente

Para evitar intentos de *hijacking*, la comunicación entre las aplicaciones (vía SDK) y el agente debe verificarse mediante la inspección de la firma de código. El agente comprobará que la *hash key* de la aplicación solicitante está registrada en el servidor mediante la invocación a un *endpoint* del servidor de autorización, y las aplicaciones comprobarán que los *Intent* que envían al agente se envían al agente correcto comprobando la *hash key* del agente, para lo cual el SDK tendrá registrada la *hash key* del mismo. Dependiendo de si es necesario integrar aplicaciones de terceros, el registro puede hacerse de dos formas:

1. Si es necesaria la integración, el registro se hará a través de un fichero de recursos llamado ***keys.xml*** que cada aplicación necesitará completar si necesita registrar una *hash key* de un agente distinto.
2. Si no es necesaria la integración la *hash key* del agente pueden estar incluida directamente en el SDK.

```
<?xml version="1.0" encoding="utf-8"?>

<resources>

    <key>klsjg899KKDuurnrr5095</key>

    <key>0949LmfjMNrn809dfjkMN</key>
```

Para evitar reiteradas comprobaciones de la misma *hash key* y mejorar con ello el rendimiento se puede usar una caché en el agente. Dado que es aconsejable que

los *Intent* para iniciar servicios sean explícitos, de nuevo se contemplan dos posibilidades en función de la necesidad de integrar aplicaciones de terceros:

1. Si es necesaria la integración, para no acoplar el SDK con el agente, lo cual dificultaría la integración, podría configurarse el nombre explícito del servicio expuesto por el agente en algún fichero de configuración (por ejemplo ***serviceToken.xml***), igual que se hace con las *hash key* de los agentes reconocidos, o bien implementar alguna interfaz definida en el SDK que proporcionase la información necesaria o cualquier otro mecanismo que permita no conocer en tiempo de compilación el nombre de la clase receptora del *Intent*.
2. Si no es necesaria la integración el nombre del *Intent* puede estar *hardcoded* en el SDK.

Para poder obtener el identificador de la aplicación será necesario usar mecanismos tipo *Binder* o *Messenger*.

```
public interface ClientContext {

         public String[] getAgentsHashedKeys();

         public String getServiceToken Name();
```

```
private String extractCertificateFromUid (int uid ) {
try {
       String packageName = context . getPackageManager (). getPackagesForUid (uid )[0];
       PackageInfo packageInfo = context . getPackageManager (). getPackageInfo ( packageName ,
PackageManager . GET_SIGNATURES );
       return packageInfo . signatures [0]. toCharsString ();
} catch ( NameNotFoundException e) {
              return null ;
       }
}
```

```
<?xml version="1.0" encoding="utf-8"?>

<resources>

    <serviceToken>es.trafico.maut.neg.service.TokenService</serviceToken>

</resources>
```

6.9.3 Asegurar la comunicación entre el agente y el servidor de autorización

Como ha quedado dicho en el punto 6.9.1, para evitar fenómenos *man-in-the-middle* el agente comprobará el certificado del servidor en las comunicaciones TLS. Adicionalmente puede comprobarse el estado de validez y no revocación del certificado invocando a los servicios corporativos de firma digital.

Por parte del servidor, es necesario identificar al agente. En OAuth suele hacerse aportando, por parte del cliente, el identificador y el secreto obtenidos durante el registro del cliente; pero dada la naturaleza pública de las aplicaciones nativas no se puede garantizar la custodia de las credenciales, luego se hace necesario un mecanismo alternativo o adicional de autenticación. Se optará por un mecanismo adicional, que puede consistir en aportar la *hash key* del cliente y su nombre de paquete base, o bien en una comunicación basada en TLS mutuo, aportando el cliente un certificado que será validado por el servidor (o ambas cosas).

6.9.4 Restricción del ámbito

Se emitirán *token* a las aplicaciones con el ámbito (*scope*) más restringido posible, asegurando que cada aplicación obtiene tan sólo lo que necesita. Tan sólo el agente recibirá un *token* primario con ámbito genérico, que el agente no usará para acceder a recursos *backend* sino tan sólo para comunicarse con el servidor de autorización y obtener *token* de ámbito restringido a través del *token* de refresco. El ámbito, pues, estará pre-asignado, lo cual es lo recomendable en escenarios *Resource Owner Password Credentials* como en el que se define en esta especificación.

6.9.5 Resource Owner Password Credentials

Este tipo de *grant*, en el que se basa la presente propuesta, implica el conocimiento por parte del cliente de las credenciales del usuario, cosa que OAuth pretende evitar.

Sin embargo, nos encontramos en el escenario en el cual este tipo de *grant* puede usarse sin apenas riesgos, debido a que las aplicaciones están controladas (son desarrolladas por el mismo organismo que las aplicaciones de *backend*) y el ámbito de los *token* está pre-establecido, con lo cual se evitan los dos riesgos de este tipo de *grant*: que las credenciales puedan perderse y que se pueda abusar del ámbito solicitado ya que no existe conocimiento del cliente del ámbito solicitado al no existir fase de autorización. El agente, que al igual que el resto de aplicaciones pasará por un control de calidad, en ningún caso persistirá las credenciales del usuario cuando las introduzca en el formulario de *login* (ni siquiera en memoria) y el ámbito de los *token* obtenidos se decidirá en el servidor. Aparte, las credenciales no las tiene realmente el cliente, simplemente el usuario las introduce en un formulario que no es propiedad del cliente (es el agente, al que hay que considerar un representante del servidor de autorización en el dispositivo) y se usa un *grant* ROPC (*Resource Owner Password Credentials*) en vez de un gran *Authorization Code* porque en el fondo no se trata de que el usuario tenga que autorizar acceso a recursos suyos, sino que los recursos son en realidad propiedad del organismo y el flujo ROPC simplifica el proceso en ese caso.

6.9.6 Phishing

Los ataques de tipo *phishing* tienen como objetivo obtener información confidencial de forma fraudulenta, por ejemplo las *password*. Es necesario que el usuario pueda verificar que la aplicación que le pide las credenciales es una aplicación autorizada y oficial. Para ello se recomienda que en el formulario de *login* que presente el agente se incluya algún identificador con el que el usuario pueda verificar que la aplicación es oficial. Ese identificador podría ser un imagen que el usuario configure al instalar la aplicación o usarse por vez primera (o bien una imagen predeterminada que todos los usuarios conozcan), y que a partir de ahí debería aparecer siempre en el formulario de *login*.

6.9.7 Proteger el endpoint de tokens contra ataques de fuerza bruta

Será necesario que el *endpoint* que el servidor de autorización expone para la obtención de *token* esté protegido contra ataques de fuerza bruta que intenten encontrar las credenciales correctas de identificación de los clientes. Para ello se tomarán medidas como el filtro de IP, la limitación de peticiones por unidad de tiempo y técnicas similares.