# Marco tecnológico para la difusión de herramientas software del sector agroalimentario: infraestructura y cliente Android.

Sergio Camarasa Bomboí
Director: Dr. Joan Fons i Cors

Valencia 2014

UNIVERSIDAD POLITECNICA DE VALENCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

# Agradecimientos

Quiero mostrar mi más sincero agradecimiento...

A mis padres, por su apoyo incondicional en todo momento y situación.

A Marlen Herrera, por su apoyo y su ánimo constantes. Sin ti no lo habría conseguido.

A mi tutor Joan Fons, por la ayuda y confianza mostradas.

Al profesor Fernando Martínez, por darme la oportunidad de emprender este proyecto.

Al profesor Miguel Ángel Jiménez, por toda la ayuda prestada con relación a los procesos agronómicos.

# Tabla de contenido

# Tabla de ilustraciones

# 1. Introducción

## 1.1. Contexto del trabajo final de máster

El presente trabajo final de máster ha sido desarrollado en el Instituto de Ingeniería del Agua y Medio Ambiente (IIAMA) (1) de la Universidad Politécnica de Valencia con la colaboración del Instituto Valenciano de Investigaciones Agrarias (IVIA) (2) y en el marco del proyecto AGADAPT (3).

Para la realización de este trabajo se han aplicado gran parte de los conocimientos adquiridos en el máster de Ingeniería del Software, Métodos Formales y Sistemas de Información (4) impartido en el Departamento de Sistemas Informáticos y Computación (DSIC) (5) de la Universidad Politécnica de Valencia (6). Han sido especialmente relevantes los conocimientos adquiridos en las siguientes asignaturas:

- Modelado, Diseño e Implementación de Servicios Web.
- Desarrollo de Sistemas Ubicuos e Inteligencia Ambiental.
- Ingeniería de la programación.
- Desarrollo de Aplicaciones en Java.
- Técnicas Avanzadas de Depuración.

## 1.2. Motivación

Es habitual que en los institutos de investigación, y en otras instituciones académicas o de otro ámbito que no sea el empresarial, se genere gran cantidad de software de muy diversa índole y muy diverso tamaño. Software que puede ir desde grandes y complejos sistemas informáticos hasta pequeños algoritmos pensados para solucionar problemas concretos.

Ocurre que muchas de estas herramientas tienen un gran potencial de cara al usuario común, pero no gozan de la visibilidad y repercusión necesarias para sacarles partido. Por ejemplo, suele darse el caso de herramientas software que pueden ser útiles para un usuario medio y que sin embargo, no han sido adaptadas para su uso fuera del ambiente académico y por lo tanto, requieren de unos conocimientos técnicos concretos para su correcta utilización.

También es habitual que estas herramientas cuenten con un grado muy bajo de usabilidad, lo que ocasiona, entre otras cosas, una pobre experiencia de uso y una mayor curva de aprendizaje de cara al usuario. Todo esto ocasiona que el producto software en cuestión no resulte atractivo para el cliente final.

Para revertir esta situación es importante cambiar el enfoque con el que se desarrolla el software. Es necesario que las aplicaciones estén centradas en la experiencia del usuario y no solo en la funcionalidad. Se deben llevar las aplicaciones a la plataforma con más auge actualmente: los dispositivos móviles. O bien desarrollar aplicaciones web con interfaces atractivas y potentes. De esta forma se consigue un resultado final más útil y atractivo para el usuario.

## 1.3. Objetivos

El objetivo principal del presente trabajo es facilitar el acceso y la visibilidad de alguna de las herramientas software desarrolladas en el IIAMA generalmente relacionadas con la optimización de los procesos de riego u otros procesos del sector agroalimentario. Para la realización de este objetivo se definirá una estructura de servicios común para todo el sistema y se implementará una aplicación cliente que haga uso de estos servicios.

La capa de servicios se debe crear considerando que futuros trabajos de desarrollo de software puedan acceder a ella y ampliarla si fuese necesario. De esta manera se pretende centralizar el código común, con todas las ventajas que ello conlleva: facilidad de mantenimiento, mejora de la reusabilidad y mayor control sobre el código, entre otras.

Otro de los objetivos del trabajo es que la herramienta desarrollada resulte fácil de utilizar por usuarios con pocos conocimientos técnicos sin que este hecho repercuta en la merma de alguna de sus funcionalidades. La aplicación debe ser igualmente útil para usuarios avanzados con un mayor conocimiento técnico del área de estudio.

## 1.4. Solución propuesta

Para alcanzar los objetivos mencionados en el apartado anterior, se ha diseñado e implementado una infraestructura con una filosofía orientada a servicios, que es capaz de dar soporte al acceso concurrente de varias aplicaciones de distinta naturaleza. En una primera fase, correspondiente al alcance de este trabajo, la capa de servicios expone la funcionalidad de una de las herramientas utilizadas en el IIAMA. Los criterios para la elección de esta herramienta se han centrado, sobre todo, en la

posibilidad de crear una aplicación atractiva para el usuario final a partir de una herramienta legada con las carencias mencionadas anteriormente: alcance reducido, poca usabilidad, etc.

La herramienta existente elegida para ser encapsulada mediante la capa de servicios web ha sido el módulo de cálculo de riego desarrollado en el IVIA y accesible desde su página web (7). Este módulo tiene un ámbito limitado al territorio de la Comunidad Valenciana y permite, habiendo indicado algunos parámetros técnicos de una parcela agrícola, predecir las necesidades hídricas del cultivo de la parcela para los próximos días. Sin embargo, esta herramienta adolece de los problemas descritos en el punto 1.2, pues carece de un diseño atractivo así como de la facilidad de uso necesaria para que sea una herramienta potencialmente accesible para usuarios con menos conocimientos técnicos.

Dentro del marco de este trabajo se contempla también la creación de una aplicación cliente que acceda, mediante la capa de servicios, a las funcionalidades de la herramienta de cálculo de riego. Se ha optado por desarrollar esta aplicación para la plataforma móvil Android (8), fundamentalmente por dos motivos: su amplia extensión en el mercado de los dispositivos móviles y por los recursos materiales disponibles para el proyecto.

La aplicación cliente se ha denominado FarmCrop y su desarrollo no se ha centrado solo en el módulo de cálculo de riego, sino que además ha sido mejorada con diversos módulos que complementan las funcionalidades de cálculo de necesidades hídricas con nuevas características, relacionadas con el ámbito agroalimentario y que pueden resultar de gran interés para el usuario final. Entre estas características destacan la definición y posicionamiento de parcelas, la generación de recomendaciones automáticas de riego o la consulta de las mediciones generadas por sondas y estaciones climáticas. Además, se han aprovechado las características de geolocalización de los dispositivos móviles para dotar en lo posible de inteligencia ambiental a la aplicación y, de esta forma, mejorar la experiencia del usuario.

## 1.5. Estructura de la memoria

El presente documento está estructurado de la siguiente forma:

- Capítulo 1: Introducción.
- Capítulo 2: Breve análisis de los diferentes métodos de cálculo de riego existentes.
- Capítulo 3: Principios tecnológicos utilizados en el proyecto, destacando

las partes especialmente importantes.

- Capítulo 4: Tareas de análisis y diseño llevadas a cabo durante la realización del proyecto.
- Capítulo 5: Implementación de los componentes más destacables del sistema.
- Capítulo 6: Conclusiones, valoración personal y trabajo futuro.
- Capítulo 7: Referencias.
- Anexo I: Bocetos de las interfaces de usuario de la aplicación FarmCrop.
- Anexo II: Manual de usuario de la aplicación FarmCrop.

# 2. Necesidades hídricas de los cultivos

En este capítulo se da una visión global de los diferentes métodos de cálculo de riego existentes, haciendo especial hincapié en el método utilizado en este proyecto (9). También se habla de la red SIAR, el sistema que hace posible la obtención automatizada de datos agroclimáticos en zonas de cultivo, para su posterior procesado.

## 2.1. Cálculo de las necesidades hídricas de un cultivo

A continuación se exponen los tres métodos existentes para calcular las necesidades hídricas de un cultivo. Cada uno de estos métodos está basado en una estrategia diferente, como son: sondeo de la humedad del suelo, consulta del estado hídrico de la propia planta y medición de parámetros climáticos (10).

### 2.1.1. Medición de la humedad del suelo

Mediante la monitorización de la humedad del suelo del cultivo se puede delimitar cuales son las necesidades de las plantas y controlar que la cantidad de agua proporcionada a las mismas es correcta. Se pretende, de esta forma, evitar pérdidas de agua en zonas profundas así como los posibles déficits hídricos ocasionados.

Existen diversos tipos de sensores que proporcionan la medida de la humedad del suelo. De reciente aparición son las sondas capacitivas FDR cuyo funcionamiento se basa en la emisión de una señal electromagnética con la que se estima la humedad, midiendo la diferencia de frecuencia entre la señal de salida y de entrada. En la aplicación desarrollada bajo este trabajo se han incluido, de forma experimental, doce de estas sondas para las que se puede consultar sus mediciones por fecha y profundidad del suelo. Como trabajo futuro, se pretende que el usuario pueda añadir sus propias sondas de humedad y dado el caso, contrastar sus resultados con los proporcionados por el cálculo del apartado 2.1.3.

**Ilustración 1: Sonda capacitiva FDR con sensores a diferentes niveles de profundidad**

Generalmente este tipo de sondas suelen estar dotadas con varios sensores que se pueden ubicar a diferentes profundidades, normalmente situadas entre los 10 y los 50 cm. Tienen un tiempo de respuesta rápido y los datos recogidos son enviados telemáticamente a un servidor.

El principal inconveniente de este tipo de sondas radica en su elevado coste, si bien se espera que su precio vaya disminuyendo. Además dado que la humedad del suelo puede variar bastante en diferentes puntos de una misma plantación, es necesaria la instalación de varias sondas para obtener datos más ajustados. Todos estos factores dificultan, de momento, la implantación habitual de estas sondas por parte del agricultor.

### 2.1.2. Medición del estado de la planta

Mientras que las otras estrategias para el cálculo del estado hídrico de las plantas se basan en métodos indirectos de medición, la estrategia tratada en este apartado consiste en medir directamente en la propia planta diversos parámetros que nos indiquen su estado hídrico.

Por un lado se puede determinar el potencial de tallo ($\Psi_{tallo}$) de una planta mediante una cámara de presión (10). Sabiendo el tipo de cultivo y contando con un conjunto de $\Psi_{tallo}$ de referencia, es posible averiguar en qué estado hídrico se encuentra la planta. Sin embargo este método cuenta con el inconveniente de que es difícilmente automatizable y requiere de personal especializado para su correcta ejecución.

Por otro lado, se puede estimar el estado hídrico de las plantas mediante la utilización de dendrómetros. Estos aparatos permiten medir las variaciones de grosor del diámetro de un tronco o rama, transformándolas en señales eléctricas. Los árboles siguen un ciclo de contracción-expansión del tronco a lo largo de un día. Estas variaciones de tamaño son diferentes en función del estado hídrico de la planta pero también se ven afectadas por las condiciones climáticas, hecho que complica la obtención de un resultado ajustado. Otro inconveniente detectado en estos sensores es que los resultados obtenidos varían mucho de una planta a otra, lo que obliga a la colocación de más sensores con el consecuente encarecimiento del equipo en conjunto.

**Ilustración 2: Dendrómetro instalado en el tronco de un naranjo (IVIA)**

### 2.1.3. Cálculo de las necesidades hídricas mediante parámetros climáticos

El último de los métodos consiste en realizar un balance hídrico midiendo la cantidad de agua evaporada del suelo y reponiéndola, si es necesario, según las necesidades específicas de cada cultivo. Este cálculo es conocido como el método FAO (11) y ha sido el utilizado en este proyecto.

La necesidad real neta de un cultivo se puede definir como la cantidad máxima de agua que puede retener un cultivo en un momento dado (12). Viene definida por la siguiente fórmula:

$$NR_n = K_c \cdot ET_o - P_e$$

$ET_o$ (evapotranspiración de referencia) se define como la cantidad de agua evaporada en una superficie y un cultivo establecidos como referencia (11). Se trata de un parámetro climático que puede ser calculado a partir de los datos obtenidos por las estaciones meteorológicas. En el caso de las estaciones del IVIA, se almacena la $ET_o$ para cada una ellas con una periodicidad diaria, para utilizarla posteriormente en el proceso de cálculo de riego.

Para cada tipo de cultivo considerado en el proceso, existe un coeficiente de cultivo ($K_c$) en el que se tienen en cuenta las características del propio cultivo, así como los efectos ponderados de la evaporación del agua del suelo. Para calcular el $K_c$ se diferencia entre cultivos hortícolas y cultivos frutales. En los primeros el coeficiente normalmente varía según la época del año y según el estado de la plantación. En los cultivos frutales, este valor suele ser fijo todo el año y solo depende de la especie frutícola.

La precipitación efectiva ($P_e$) es la cantidad de precipitación real aprovechada por la planta, dado que parte de la precipitación se pierde, fundamentalmente por escorrentía. Para su cálculo se usan normalmente fórmulas empíricas adaptadas a las características de la zona geográfica. Por ejemplo en el IVIA, y por extensión en este proyecto, se utiliza la siguiente fórmula para el cálculo de la precipitación efectiva aplicada a los cítricos:

$$P_e = 1.25 \cdot \frac{\pi \cdot D_a^2}{4 \cdot a \cdot b} \cdot P_d$$

Donde $a$ y $b$ son la distancia entre plantas y entre líneas de plantación respectivamente, $P_d$ la precipitación diaria y $D_a$ el diámetro de copa de la planta.

Se debe tener en cuenta que las necesidades reales netas ($NR_n$) corresponden a la cantidad de agua que necesita absorber una planta, pero que no toda el agua suministrada puede ser absorbida por la misma. Se debe entonces calcular las necesidades totales ($NT_R$) de la planta para asegurar que recibe como mínimo una cantidad de agua igual a sus $NR_n$ (12). Los motivos por los que se deben corregir las $NR_n$ son fundamentalmente tres.

- Salinidad en el agua de riego. En este caso se suele añadir una porción de agua al proceso de riego, denominada fracción de lavado, para reducir la salinidad alrededor de la planta.

- Pérdidas de agua por percolación profunda. Es decir, la cantidad de agua que se filtra más allá de la zona de la raíz.
- Uniformidad de emisión. Debido a que no todos los emisores emiten la misma cantidad de agua, es necesario aumentar el volumen de riego para asegurar que cada planta recibe al menos sus $NR_n$.

Una vez determinadas las necesidades totales, podemos calcular el tiempo de riego necesario ($T_{rieg}$) para suministrar esta cantidad a cada una de las plantas a través de la siguiente ecuación:

$$T_{Rieg} = \frac{NT_R \cdot a \cdot b}{n_{emis} \cdot q_{emis}}$$

Donde $n_{emis}$ representa el número de emisores por planta y $q_{emis}$ el caudal medio de los emisores.

## 2.2. Sistema de Información Agroclimática para el Regadío (SIAR)

En el marco del proyecto europeo INTERREG II-C (13) de lucha contra la sequía, el Ministerio de Agricultura, Alimentación y Medio Ambiente instaló durante los años 1998 a 2001 un Sistema de Información Agroclimática para el Regadío (SIAR) (14) con el objetivo de promover el ahorro de agua, proporcionando al agricultor información de base suficiente para calcular las necesidades hídricas de sus cultivos.

La red SIAR consta de unas 350 estaciones agroclimáticas automáticas repartidas por todo el territorio nacional, aunque con más incidencia en las comunidades autónomas deficitarias de agua. Siguiendo una estructura jerárquica, los datos recogidos por estas estaciones son procesados en 12 Centros Zonales y la información resultante se envía a un Centro Nacional que actúa como coordinador de la red. Es en estos centros donde se realizan los cálculos pertinentes para obtener las necesidades hídricas a partir de los datos recogidos por las estaciones de la red. En concreto, en la Comunidad Valenciana la red SIAR cuenta con 55 estaciones agroclimáticas repartidas por aquellas zonas que cuentan con cultivos bajo riego.

**Ilustración 3: Estación agroclimática de Moncada (IVIA)**

Los parámetros climáticos que transmiten estas estaciones y que son necesarios para calcular las necesidades de riego son: temperatura y humedad del aire, velocidad y dirección del viento, radiación solar y precipitación. Además, en cada centro zonal cuentan con información específica de cada zona de regadío que gestionan, aumentando de esta forma la exactitud de los cálculos (2).

# 3. Tecnologías

En este capítulo se proporciona una visión global de las tecnologías más representativas utilizadas en este proyecto, resaltando aquellas partes principalmente implicadas en el desarrollo del mismo.

## 3.1. Android

Android es un sistema operativo de código abierto, basado en el *kernel* de Linux y diseñado especialmente para dispositivos móviles (8). Fue desarrollado inicialmente por Android Inc., compañía que posteriormente sería adquirida por Google. Junto con el anuncio del lanzamiento del sistema Android, Google lideró la creación de la *Open Handset Alliance* (15), un grupo de compañías de hardware, software y telecomunicaciones dedicadas al desarrollo de estándares abiertos para dispositivos móviles.

Por encima del núcleo Linux, Android cuenta con una capa de software formada por un conjunto de librerías escritas en C, entre las que se incluyen el administrador de interfaz gráfica, una base de datos relacional SQLite (16), la API gráfica OpenGL ES, el motor web WebKit, SGL, SSL y la biblioteca estándar de C Bionic. Sobre esta capa Android monta un framework de aplicaciones Java que son ejecutadas a través de la máquina virtual Dalvik (17).

Android cuenta con una gran comunidad activa de desarrolladores. Hasta la fecha se han contabilizado más de un millón de aplicaciones en Google Play (18) la tienda oficial online de Google; gran parte de estas aplicaciones son gratuitas. Se han publicado varias versiones del sistema operativo desde la fecha de su lanzamiento. La más reciente es KitKat (v4.4), publicada en noviembre de 2013.

La aplicación desarrollada en el marco de este proyecto está compilada con el nivel 19 de la API, perteneciente a la versión 4.4 KitKat. No obstante, en la implementación se ha intentado mantener la compatibilidad con versiones anteriores de Android. De esta forma, la aplicación es compatible con todos los dispositivos Android hasta un nivel mínimo de API 8 (v2.2 Froyo).

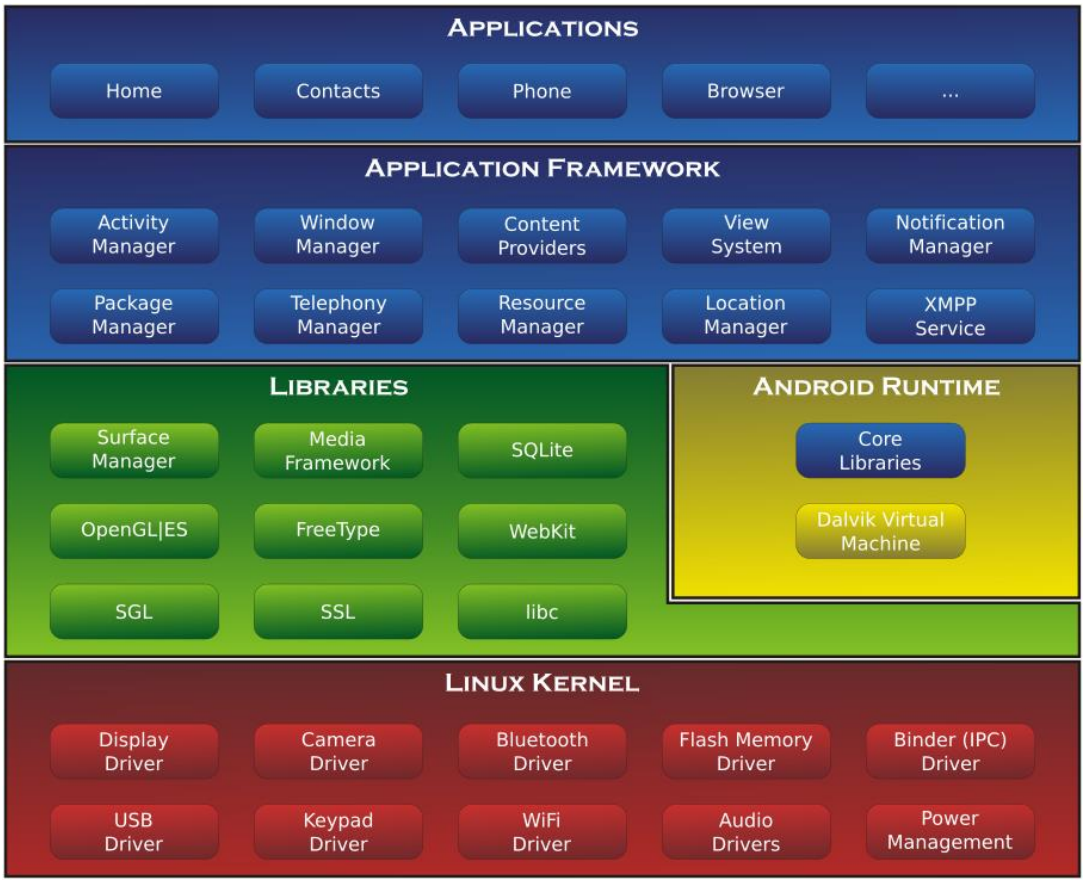



**Ilustración 4: Arquitectura del sistema Android**

### 3.1.1. Fundamentos de una aplicación Android

Una aplicación Android se compila mediante las herramientas del SDK, generando un archivo APK con todo el contenido de la misma. Este archivo es todo lo que necesita un dispositivo Android para instalar la aplicación.

Cada aplicación instalada permanece aislada del resto gracias a los diversos mecanismos de seguridad del sistema:

- Android asigna a cada aplicación un usuario diferente con un identificador propio. El sistema establece permisos a todos los archivos de una aplicación de forma que solo el usuario asignado a la aplicación puede acceder.
- Cada aplicación se ejecuta en su propio proceso de Linux.
- A su vez, cada proceso se ejecuta en su propia máquina virtual, aislado del resto del sistema.

### 3.1.2. Componentes de una aplicación

A continuación se describen los tres componentes fundamentales de una aplicación Android. Cada uno de estos componentes sirve a un propósito diferente y cuenta con un ciclo de vida y un comportamiento diferente al resto. También se habla de los *Intents*, objetos que facilitan la comunicación entre estos componentes.

#### *3.1.2.1. Activities*

Una *activity* es el componente de la aplicación que proporciona al usuario una ventana con una interfaz gráfica con la que interactuar. Generalmente esta ventana ocupa toda la pantalla del dispositivo, pero también puede ser más pequeña y mostrarse sobre otra actividad.

Normalmente una aplicación está formada por múltiples actividades independientes entre sí. Cada actividad tiene la facultad de iniciar nuevas actividades. Cuando esto sucede, la actividad antigua cesa su ejecución y el sistema la almacena en una pila (*back stack*). Entonces la nueva actividad lanzada se muestra en primer plano. Cuando el usuario termina sus acciones con la nueva actividad y pulsa el botón *Back*, la actividad anterior es recuperada de la pila y mostrada de nuevo al usuario.

La interfaz de usuario de una actividad viene definida por un conjunto de vistas (objetos que heredan de la clase *View*). Cada una de estas vistas controla un rectángulo determinado de la ventana y tiene la capacidad de responder a las acciones del usuario.

Se pueden diferenciar dos tipos de vistas según su funcionalidad aportada. Por un lado, los *widgets* proporcionan elementos visuales e interactivos a la pantalla. Es el caso de los botones, los campos de texto o los *checkbox*. Por otro lado, los *layouts* son vistas que contienen otras vistas a las que dotan de un modelo distributivo determinado. En ambos casos, el desarrollador puede heredar estas clases para crear sus propias vistas personalizadas.

Es posible crear una vista de forma procedural, en el mismo código, o bien de forma declarativa a través de un fichero XML que forme parte de los recursos de la aplicación. Sin embargo, el uso de este último método permite mantener separado el diseño de la interfaz del comportamiento de la actividad descrito en el código.

Un factor imprescindible a la hora de diseñar actividades es controlar adecuadamente su ciclo de vida. En un instante determinado, una actividad puede encontrarse básicamente en uno de estos tres estados:

- *Resumed*: la actividad se encuentra en ejecución y es visible para el usuario.
- *Paused*: otra actividad se encuentra en primer plano pero no oculta totalmente a esta actividad, que continúa siendo visible parcialmente. Esta actividad mantiene su estado y sus datos en memoria, pero puede ser eliminada por el sistema en caso de necesitar más memoria.
- *Stopped*: la actividad se encuentra en segundo plano y no es visible para el usuario. Al igual que en el estado anterior, esta actividad se mantiene activa en memoria pero puede ser eliminada si el sistema necesita más memoria.

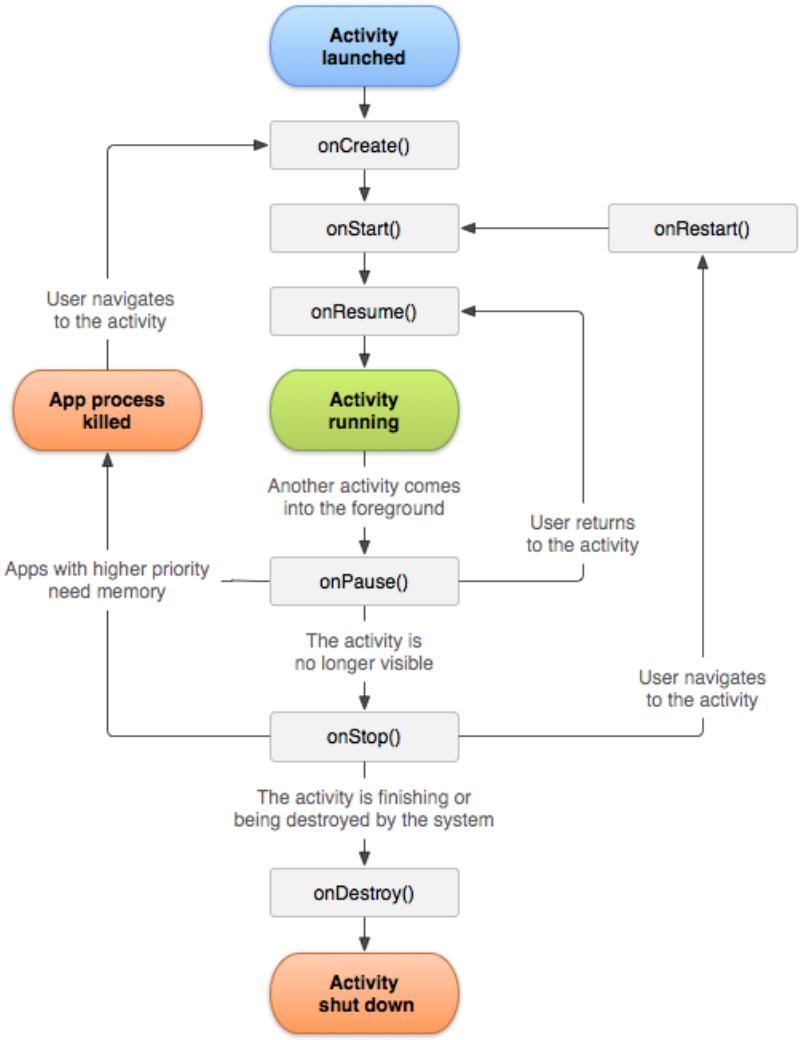



**Ilustración 5: Ciclo de vida de las actividades**

Cuando una actividad pasa de un estado a otro, el sistema lo notifica a través de una serie de métodos de *callback*. El desarrollador puede sobrescribir estos métodos para proporcionar el comportamiento adecuado a la actividad en función del cambio de estado producido. En la Ilustración 5 se muestra el ciclo de vida completo de una actividad, junto con los métodos de *callback* que lanza el sistema en cada transición.

El sistema permite almacenar los datos cruciales de una actividad para que, en el caso de que ésta sea destruida, pueda ser recreada en el mismo estado en la que se encontraba. Este proceso se lleva a cabo de forma transparente para el usuario, que acaba viendo la actividad exactamente como la dejó. Para implementar este mecanismo, el desarrollador cuenta con dos métodos *callback*, uno para almacenar la información y otro para recuperarla. En la Ilustración 6 se muestra este proceso en detalle.

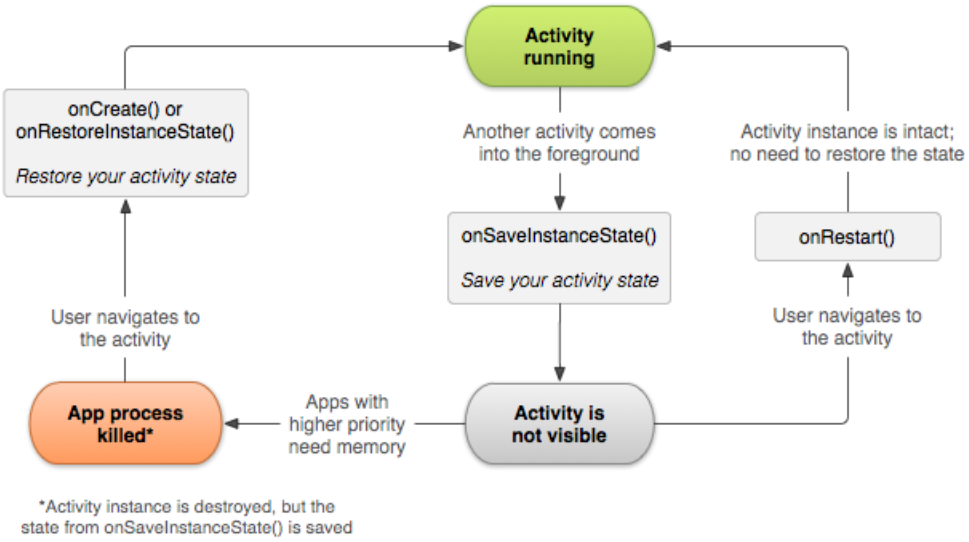



**Ilustración 6: Proceso de restauración de las actividades**

Cuando Google lanzó Android 3.0 en el 2011, introdujo un nuevo e importante elemento para el diseño de las interfaces gráficas: los *fragments*. Se puede definir este elemento como una sección modular de una actividad, que puede ser reutilizable y que dispone de su propia interfaz de usuario, su propio ciclo de vida y sus propios eventos. Los *fragments*, además, pueden ser añadidos o eliminados de una actividad mientras ésta está en ejecución.

El uso de *fragments* otorga al desarrollador una mayor flexibilidad, puesto que le permite cambiar la apariencia de una actividad en tiempo de ejecución. Los cambios realizados (en forma de *fragments*) son almacenados en una *back stack* propia de la

actividad y pueden ser restaurados posteriormente en un proceso similar al llevado a cabo por las actividades.

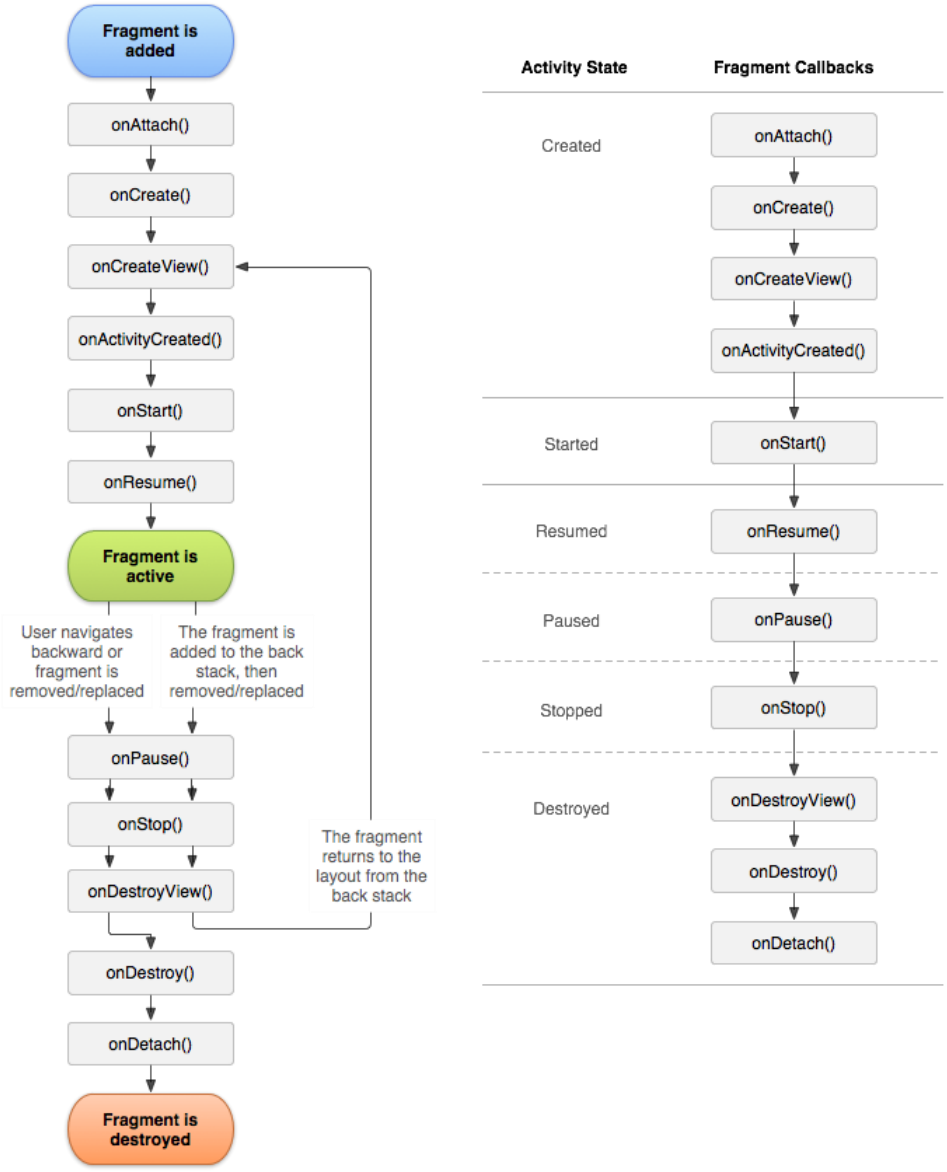



**Ilustración 7: Ciclo de vida de los *fragments* y comparación con el de las actividades**

### *3.1.2.2. Services*

Un *service* es un componente de una aplicación Android carente de interfaz, que está capacitado para realizar operaciones de larga duración en segundo plano. Una aplicación puede lanzar servicios, que se mantendrán en ejecución aunque el usuario cambie a otra aplicación. Estos componentes son útiles a la hora de realizar

transacciones a través de la red, realizar operaciones de I/O, reproducir música y en general cualquier acción que pueda beneficiarse de una ejecución en segundo plano.

Un servicio puede existir en dos formas básicas que en la práctica pueden coexistir:

- "Iniciado": el servicio es lanzado por algún componente de alguna aplicación. A no ser que se especifique explícitamente, el servicio continuará en ejecución aunque el componente que lo lanzó sea destruido.
- "Enlazado": el servicio está enlazado con al menos un componente. Se comunica con los elementos que lo enlazan mediante una interfaz y estará en ejecución mientras existan estos enlaces. En el momento en que ningún componente esté vinculado al servicio, éste será eliminado del sistema.

#### *3.1.2.3. Content Providers*

Los *content providers* son componentes de una aplicación Android cuya función es la de gestionar el acceso a un conjunto ordenado de datos. Proporcionan una interfaz estándar que conecta los datos en un proceso con otro elemento que está ejecutándose en un proceso diferente.

Este tipo de componentes solo necesitan ser desarrollados cuando se desee compartir información compleja con otras aplicaciones. Android proporciona una serie de *content providers* ya desarrollados para acceder a datos de imágenes, videos, música o contactos personales. Cualquier aplicación puede acceder a estos *providers* del sistema.

Un *content provider* presenta los datos a sus aplicaciones cliente en forma de tabla, de forma similar a una base de datos relacional. Cada fila representa una instancia de un tipo de datos en particular. Las aplicaciones acceden a los *content providers* mediante un objeto del sistema que proporciona métodos con las operaciones básicas de manipulación de datos (creación, selección, modificación y borrado).

#### *3.1.2.4. Intents*

Los *intents* son objetos cuya función es la de transmitir mensajes entre los diferentes componentes de las aplicaciones, con el fin de facilitar la comunicación

entre éstos. Generalmente se usan para iniciar actividades o servicios, o bien para enviar una notificación en todo el sistema (*broadcast*).

Android define dos tipos de *intents*:

- *Intents* explícitos: especifican directamente el componente que quieren iniciar referenciándolo por su nombre. Este es el caso típico cuando una aplicación desea iniciar una actividad propia, pues conoce exactamente el nombre del componente (la nueva actividad en este caso).
- *Intents* implícitos: no se menciona ningún componente específico cuando se crean, en su lugar, indican una acción genérica a realizar. La petición llegará a otro componente que esté configurado para gestionar esta acción.

Cuando se crea un *intent* explícito, Android inmediatamente inicia el componente especificado. En cambio, cuando se crea un *intent* implícito, el sistema busca que componentes son capaces de gestionarlo y en el caso de encontrar alguno, lo inicia y le envía el *intent* inicial (Ilustración 8). En el caso de que existan múltiples elementos capaces de gestionar la acción requerida, el sistema muestra un diálogo para que sea el usuario quien elija que componente utilizar.

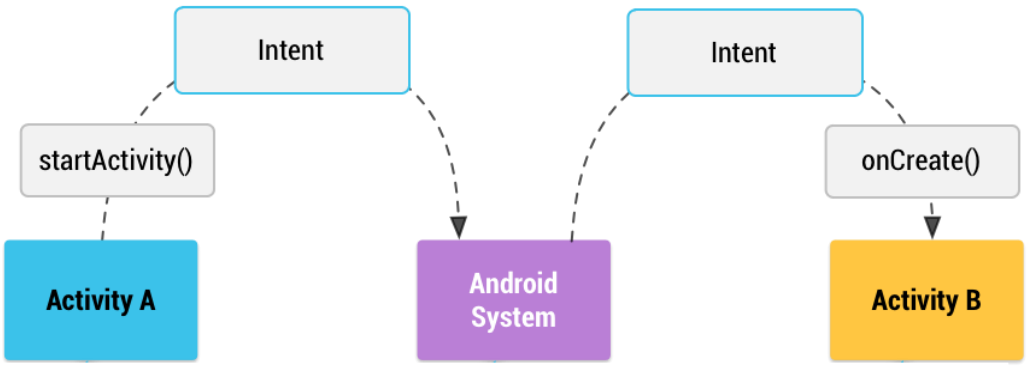



**Ilustración 8: Inicio de una nueva actividad mediante un *intent* implícito**

Un *intent* implícito permite iniciar una actividad en otra aplicación solo definiendo la acción a realizar y adjuntando los datos derivados de esta acción (si los hubiese). Android dispone de varios *intents* implícitos que el desarrollador puede utilizar para extender su aplicación. A continuación se muestran las funcionalidades más relevantes a las que dan acceso:

- Alarmas: crear una alarma, crear un temporizador, mostrar todas las alarmas.
- Calendario: añadir un evento al calendario.

- Cámara: capturar una imagen o un vídeo.
- Contactos: seleccionar un contacto, ver un contacto, editar un contacto, añadir un nuevo contacto.
- Email: componer un email a partir de los datos proporcionados.
- Almacenamiento de archivos: abrir un archivo.
- Mapas: mostrar una ubicación en el mapa.
- Multimedia: reproducir un archivo multimedia, reproducir una lista de canciones.
- Teléfono: iniciar una llamada.
- Configuración: abrir una sección específica de la configuración del dispositivo.
- Mensajes de texto: componer un mensaje de texto a partir de los datos proporcionados.
- Navegador web: abrir una página web, realizar una búsqueda.

### 3.1.3. Recursos de la aplicación

Android facilita al desarrollador la tarea de mantener la independencia entre el código y los recursos de la aplicación (imágenes, cadenas de texto, etc.). Mantener esta separación permite definir recursos personalizados para distintos tipos de dispositivos sin cambiar el código de la aplicación.

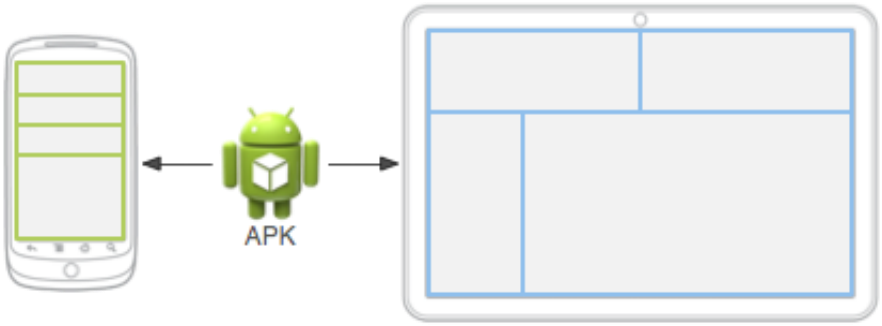



**Ilustración 9: Dos dispositivos diferentes con recursos adaptados**

El sistema utiliza una serie de carpetas, con una nomenclatura específica, para organizar los recursos según su tipo y configuración. Para cada tipo de recurso se

pueden definir recursos por defecto, o bien, recursos específicos para una configuración en concreto. En la Ilustración 9 se puede apreciar como para una misma aplicación, el sistema aplica recursos diferentes (*layouts* en este caso) para dispositivos con tamaños de ventana diferentes.

Cuando se compila una aplicación, se crea la clase especial *R* donde se generan tantos identificadores como recursos existan en la carpeta *res* del proyecto. Estos *IDs* se crean como enteros únicos y en forma de campos estáticos de la clase. El desarrollador puede entonces utilizar estos *IDs* para referenciar los recursos tanto en el código como en los archivos XML de la aplicación.

Los recursos que pueden ser definidos y externalizados en una aplicación Android son: animaciones, colores, imágenes, *layouts*, menús de aplicación, cadenas de caracteres, estilos y valores como booleanos, enteros, dimensiones, etc. Cada uno de estos tipos de recurso dispone de su subdirectorio correspondiente en la carpeta *res* del proyecto.

### *3.1.4.* Fichero *Manifest*

Cada aplicación Android debe tener, en su directorio raíz, un fichero XML llamado "*AndroidManifest.xml*". Este fichero contiene información esencial que el sistema necesita conocer antes de ejecutar la aplicación. Las funciones principales del *manifest* son las siguientes:

- Proporciona un nombre único de paquete para la aplicación.
- Describe los componentes de la aplicación. Para cada componente nombra la clase que lo implementa y describe sus capacidades.
- Determina el proceso que hospedará a cada componente.
- Declara los permisos que debe tener la aplicación.
- Declara el nivel mínimo de la API de Android que la aplicación necesita.
- Proporciona una lista de las librerías con las que la aplicación debe estar enlazada.

Cada vez que se crea un nuevo proyecto Android, se obtiene un archivo *manifest* inicial que se genera automáticamente, y que abarca los elementos básicos de una aplicación sencilla; la declaración de una actividad principal y diversos parámetros de configuración básicos.

### 3.1.5. Interfaz de usuario

Todos los elementos de una interfaz de usuario Android son construidos a partir de los objetos *View* y *ViewGroup*. Un objeto *View* se encarga de representar gráficamente un determinado rectángulo de la pantalla con el que el usuario puede interactuar. Los objetos *ViewGroup* actúan como contenedores de *Views* y de otros *ViewGroups* formando una estructura jerárquica y definiendo la disposición en la pantalla de los elementos de la interfaz.

Android proporciona una colección de elementos ya implementados, derivados de los objetos *View* y *ViewGroup*, que ofrecen al desarrollador controles de usuario de uso común, así como varios modelos que definen la disposición de los elementos en la interfaz.

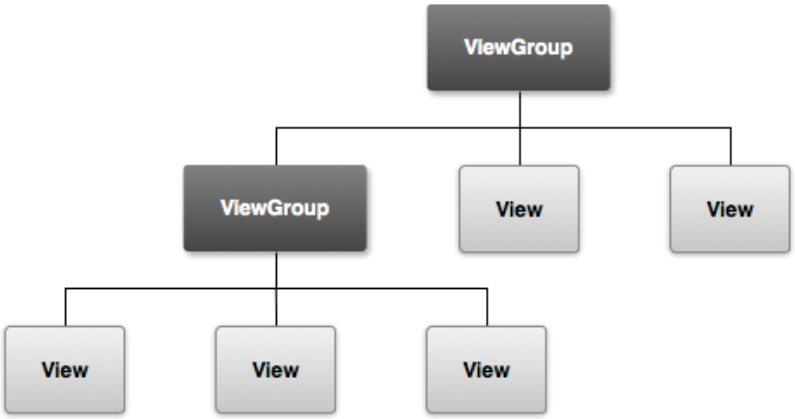



**Ilustración 10: Jerarquía de objetos de una interfaz de usuario Android**

Las estructuras jerárquicas que definen la interfaz de usuario (*layouts*) pueden ser declaradas en ficheros XML en forma de recurso de la aplicación, o bien directamente en el código, donde se instanciarán los elementos de la interfaz en tiempo de ejecución. Sin embargo es preferible que la declaración de la interfaz se realice mediante ficheros XML, ya que de este modo se consigue mejorar la independencia entre la parte de presentación de la aplicación y el código que controla el comportamiento de la misma.

Cada subclase de *ViewGroup* proporcionada por el sistema ofrece un modelo distinto de distribución de los elementos de la interfaz. Este modelo se aplica únicamente a los *Views* y *ViewGroup* que contiene anidados. Entre los ejemplos más representativos de subclases de *ViewGroup* se encuentran los *LinearLayout* y los *RelativeLayout*.

Cuando el contenido de un *layout* es dinámico o no se conoce inicialmente, el desarrollador puede hacer uso de alguna de las subclases de *AdapterView* (como

*ListView*, *ExpandableListView*, *GridView*, etc.) para generar los *views* correspondientes en tiempo de ejecución. Una subclase de *AdapterView* necesita utilizar un objeto de tipo *Adapter* para enlazar los datos con los elementos de la interfaz. El objeto *Adapter* actúa como intermediario entre el origen de datos y el *layout* del *AdapterView*.

## 3.2. .NET

.NET (19) es un framework de desarrollo de software desarrollado por Microsoft que da soporte a todo el ciclo de desarrollo, permitiendo la creación rápida de aplicaciones. Centrado sobre todo en Windows, cuenta con un gran número de librerías y diversos lenguajes de programación con la capacidad de interoperar entre ellos. La Biblioteca de Clases Base (BCL) de .NET da soporte a la mayoría de las operaciones básicas involucradas en el desarrollo de aplicaciones, incluyendo módulos de acceso a base de datos, interfaces de usuario, desarrollo de aplicaciones web, criptografía, operaciones matemáticas, comunicaciones de red, etc.

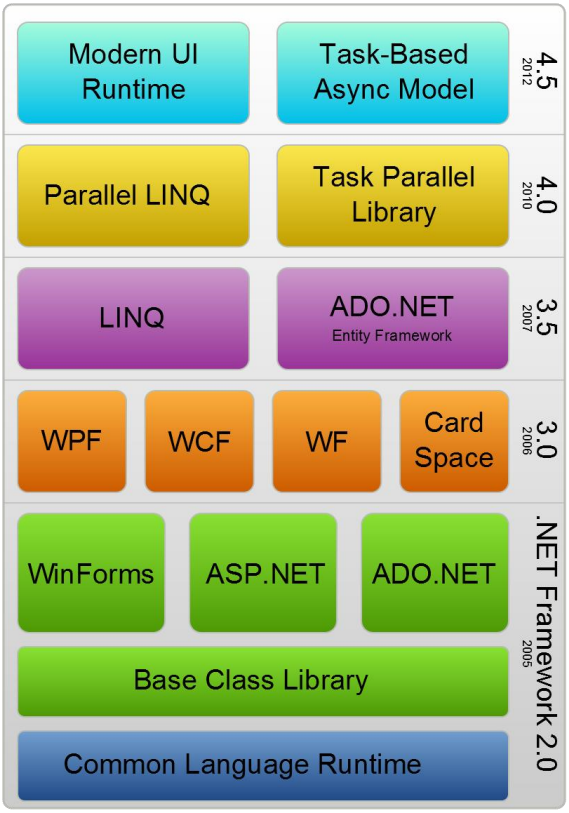



**Ilustración 11: Versiones del framework .NET y sus funcionalidades**

Las aplicaciones desarrolladas bajo la plataforma .NET se ejecutan sobre la *Common Language Runtime* (CLR), una máquina virtual de aplicaciones que proporciona servicios básicos como gestión de memoria, tratamiento de excepciones o

seguridad del código. Junto con .NET, Microsoft también proporciona un entorno integrado de desarrollo (IDE) denominado Visual Studio (20) con multitud de herramientas y funcionalidades para todas las fases del desarrollo.

A lo largo de estos últimos años Microsoft ha ido evolucionando el framework de .NET desde que viera la luz en febrero de 2002, añadiendo multitud de nuevas funcionalidades tan interesantes como *Entity Framework* o LINQ. En este proyecto se ha utilizado la versión 4.0 del framework. Esta versión tiene todas las herramientas necesarias para el desarrollo del proyecto y es la primera que incluye Web API, la extensión de ASP.NET para el desarrollo de servicios sobre HTTP.

### *3.2.1. Common Language Infrastructure*

La *Common Language Infrastructure* (CLI) es una especificación abierta desarrollada por Microsoft y estandarizada por ISO (21) y ECMA (22). Describe un entorno virtual de ejecución donde distintos lenguajes de alto nivel pueden ser ejecutados en distintas plataformas, sin que sea necesario modificar el código o recompilarlo. El framework .NET es la implementación de Microsoft de su CLI. Otras implementaciones son Portable.NET o el proyecto Mono (23).

La especificación del CLI describe principalmente cuatro aspectos:

- *Common Type System* (CTS): un sistema unificado de tipos, común a todos los lenguajes soportados por el framework.
- Metadatos: información sobre la estructura del programa, independiente del lenguaje con el que esté programado.
- *Common Language Specification* (CLS): un conjunto de normas que todo lenguaje del framework debe cumplir para poder interactuar con los otros lenguajes.
- *Virtual Execution System* (VES): un sistema de ejecución virtual capaz de combinar fracciones de código de diferentes lenguajes del framework en tiempo de ejecución. El *Common Language Runtime* (CLR) mencionado anteriormente es la implementación de Microsoft del sistema de ejecución virtual.

Todos los lenguajes compatibles se compilan a un lenguaje intermedio, independiente de la plataforma hardware, denominado *Common Intermediate Language* (CIL). El sistema de ejecución virtual, que sí es específico para cada plataforma, compila entonces el código CIL en el lenguaje máquina de cada plataforma en cuestión.

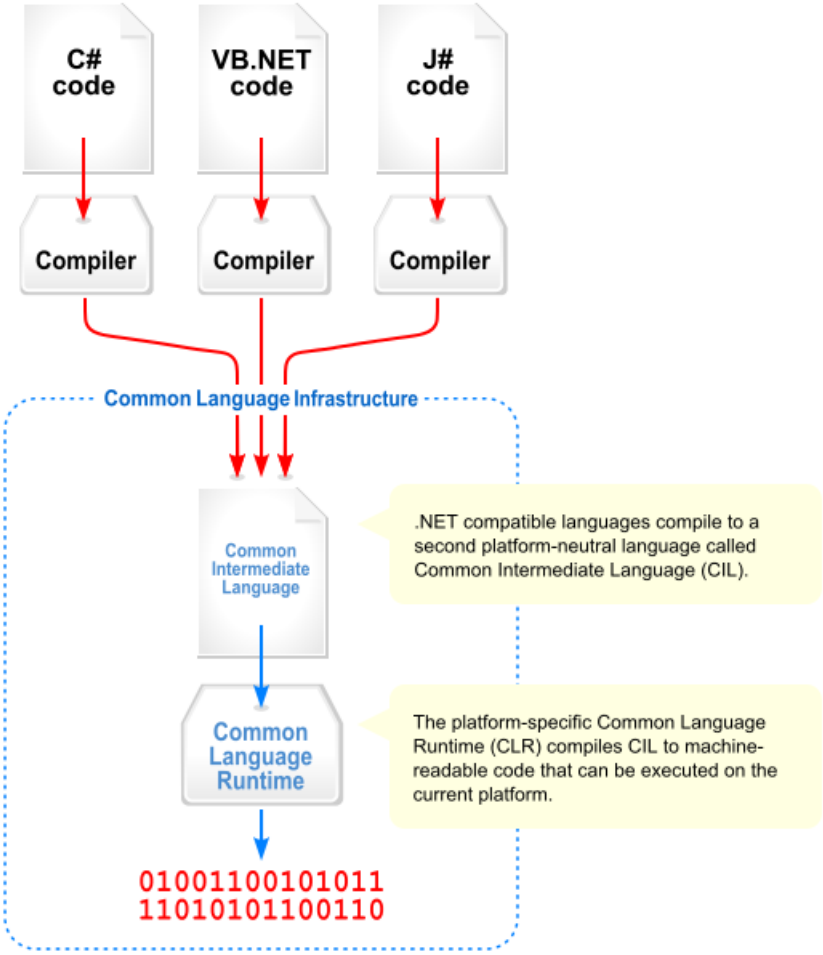



**Ilustración 12:** ***Common Language Infrastructure***

### 3.2.2. ASP.NET

ASP.NET (24) es un framework de desarrollo web diseñado para dar soporte a la creación de páginas web dinámicas, aplicaciones web o servicios web. Se creó como sucesor de *Active Server Pages* (ASP) y su primera versión se lanzó junto con el framework 1.0 de .NET.

Su principal bloque de construcción son los *web forms*, páginas web con extensión ".aspx" que combinan las etiquetas estáticas habituales de HTML con otras donde se definen controles de usuario y controles de la parte del servidor. Es en estas últimas etiquetas donde los desarrolladores colocan todo el código requerido por la página web. En la versión 2.0 del framework se introdujo el modelo *code-behind*, que permite separar el código dinámico de la página en un fichero independiente nombrado igual que la página pero con una extensión diferente.

Según ha ido evolucionando ASP.NET se han ido desarrollando nuevos modelos de programación, con metodologías de desarrollo diferentes y estructuras distintas de la aplicación. Cada uno de estos modelos posee puntos fuertes y débiles, y es responsabilidad del desarrollador escoger el que mejor se adapte a cada caso:

- ASP.NET Web Forms: fue el primero de los tres modelos. Proporciona un estilo de programación basado en eventos y controles de usuario.
- ASP.NET MVC: modelo de programación basado en el patrón de diseño MVC (modelo-vista-controlador) que proporciona una separación más clara entre las capas de interoperabilidad de la aplicación.
- ASP.NET Web Pages: este modelo ofrece al desarrollador una forma más sencilla de implementar aplicaciones web sin renunciar a la potencia y flexibilidad de ASP.NET. Es el más reciente de los tres modelos.

Microsoft ha publicado algunas extensiones que amplían las funcionalidades de ASP.NET en diversos aspectos. Algunas de las más significativas son: ASP.NET AJAX, ASP.NET MVC, ASP.NET Dynamic Data o ASP.NET Web API. Esta última extensión ha sido utilizada en este proyecto para la implementación de la capa de servicios web y se explica con más detalle en el siguiente apartado.

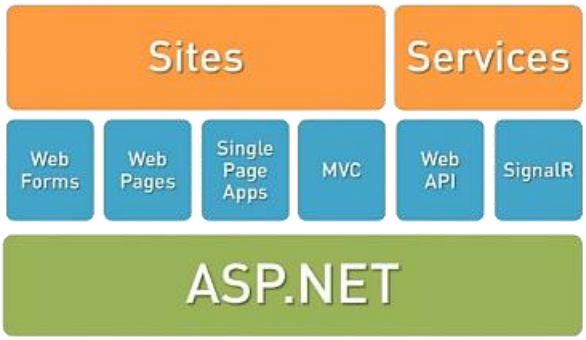



**Ilustración 13: Pila de ASP.NET**

### *3.2.2.1. ASP.NET Web API*

Web API es una extensión de ASP.NET para la creación de servicios HTTP. Es especialmente interesante su uso para el desarrollo de servicios RESTful, pues está desarrollada para que sea sensible a la semántica de los mensajes del protocolo HTTP. También comparte muchos conceptos con la extensión MVC; tanto es así que la forma de trabajar con ambas extensiones es casi idéntica. Se caracteriza también por proporcionar sencillos mecanismos para tratar las peticiones de entrada y para servir

el contenido de respuesta en el formato especificado. Web API apareció por primera vez junto con el lanzamiento de ASP.NET MVC 4.

Las principales características a las que da soporte ASP.NET Web API son las siguientes:

- Robusto mecanismo para la creación dinámica de direcciones URL.
- Formato del contenido de respuesta sensible a las cabeceras "*Accept*" de la petición.
- Soporte a los formatos de intercambio de datos más conocidos: XML, JSON, ATOM.
- Soporte por defecto de las operaciones básicas de los servicios REST (POST, GET, PUT y DELETE).
- Modelo de *Formatters* y *Filters* basado en ASP.NET MVC.
- *Self-host*: las aplicaciones realizadas con Web API se hospedan en el servidor en su propio proceso independiente. No necesitan estar embebidas en una aplicación web.

Existen una serie de escenarios donde el uso de Web API es recomendable frente a otras extensiones de ASP.NET. A continuación se citan tres de los más relevantes:

- Cuando se desea crear una API auto descriptiva de servicios HTTP para ser consumida directamente desde la web.
- En el caso de aplicaciones tipo SPA (*Single Page Applications*) o de aplicaciones con una fuerte carga de llamadas asíncronas. Es el caso de la aplicación Android tratada en este trabajo.
- Cuando se desea crear una capa de servicios HTTP genérica, para que sea consumida por varias aplicaciones. El presente trabajo podría evolucionar hasta este escenario si se decidiese, por ejemplo, implementar una página web con una funcionalidad similar a la aplicación móvil. En ese caso, ambas aplicaciones accederían a la misma capa de servicios HTTP.

## 3.3. SQL Server

Microsoft SQL Server es un sistema de gestión de bases de datos relacionales (SGBD). Tiene varias ediciones, centradas en dar soporte a perfiles de usuario y contextos diferentes. Utiliza los lenguajes de consulta T-SQL y ANSI SQL. La primera versión de SQL Server apareció en 1989 y la más reciente fue publicada en marzo de 2014. En este proyecto se ha utilizado la versión 11.0 (SQL Server 2012).

Entre las diferentes ediciones que tiene en la actualidad Microsoft SQL Server, destacan las siguientes:

- Enterprise: cuenta con todas las características completas de SQL Server y está pensado para grandes sistemas que necesiten un alto nivel de escalabilidad.
- Standard: esta edición difiere de la Enterprise en que soporta menos conexiones simultáneas y no incluye todas las características de alta disponibilidad.
- Express: edición reducida y gratuita de SQL Server. Está limitada al uso de un solo procesador, 1GB de memoria y 10GB de almacenamiento.
- Azure: SQL Azure es la versión de SQL Server de computación en la nube. Forma parte del conjunto de servicios de la plataforma Microsoft Azure.
- Compact (SQL CE): edición compacta de SQL Server. Debido a su pequeño tamaño, contiene un conjunto reducido de características comparado con otras ediciones. El tamaño de la base de datos está limitado a 4GB y debe estar embebida en la aplicación que la usa.
- Developer: esta edición contiene las mismas características que la edición Enterprise, pero su licencia está limitada a tareas de desarrollo y pruebas. Esta edición está disponible de forma gratuita para estudiantes en el programa DreamSpark de Microsoft.

### 3.3.1. Tratamiento de datos geoespaciales

Microsoft introdujo las funcionalidades necesarias para trabajar con datos espaciales en la versión 10.0 de SQL Server (SQL Server 2008). Estas nuevas características se incluyen de serie en todas las ediciones. Otros SGBD, sin embargo, necesitan de un componente adicional para proporcionar funcionalidades similares.

Los datos espaciales se utilizan para describir la posición, forma y orientación de los objetos en el espacio. Es especialmente importante situar estos objetos sobre la superficie de la tierra; se habla entonces de datos geoespaciales. En este proyecto se han utilizado datos geoespaciales para situar geográficamente parcelas, estaciones y sondas. En el caso de las parcelas, la aplicación permite definir, además, el perímetro de la misma en forma de polígono. En la Ilustración 14 se muestra la representación gráfica de los datos obtenidos por una consulta con datos geoespaciales, realizada en el SQL Server Management Studio. En concreto, estos datos corresponden a los perímetros de algunas de las parcelas definidas en la aplicación.

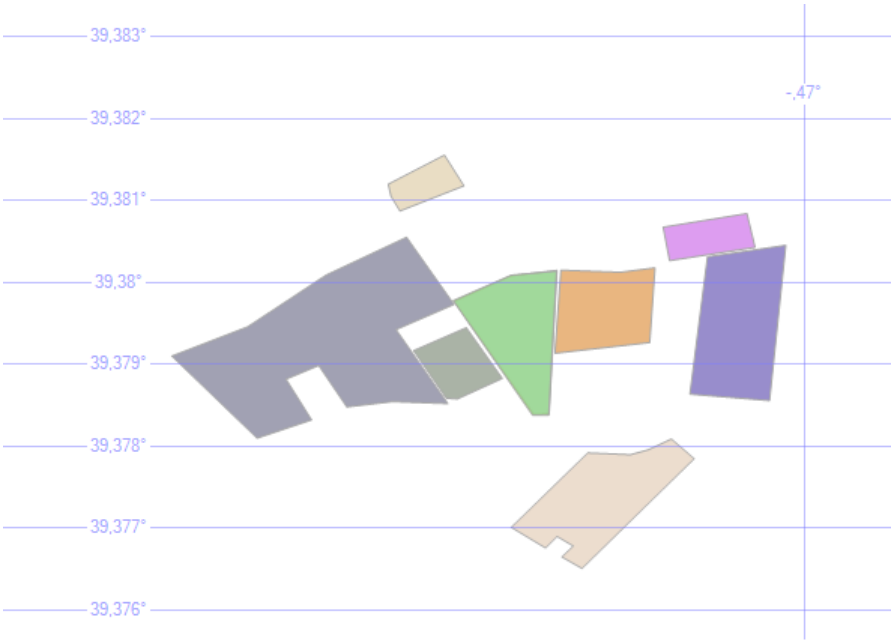



**Ilustración 14: Representación gráfica de datos geoespaciales**

En SQL Server se pueden utilizar tres tipos diferentes de geometrías para representar objetos geoespaciales: puntos, polilíneas y polígonos. Cada una de estas geometrías se define en el sistema mediante la especificación de las coordenadas de cada uno de sus vértices. Además, esta definición debe estar acompañada del identificador del sistema de referencia espacial utilizado.

Los sistemas de referencia espacial hacen posible la definición singular de puntos sobre la superficie de la tierra. En este proyecto se ha usado el sistema WGS 84 (identificador 4326), uno de los más comunes y que es utilizado, por ejemplo, por los sistemas GPS. A continuación se describen los componentes que forman parte de un sistema de referencia espacial:

- Sistema de coordenadas: define un marco de trabajo matemático para determinar la posición de un elemento en relación al origen especificado.

- Datum: representa un modelo de la tierra en el cual se puede aplicar el sistema de coordenadas.
- Primer meridiano: eje desde donde se medirán las coordenadas de longitud.
- Proyección: conjunto de parámetros requeridos para crear una imagen en dos dimensiones de la superficie de la tierra.
- Unidad de medida: unidad en la que están expresadas las coordenadas.

En SQL Server existen dos tipos de datos espaciales. Por un lado el tipo *geography* se utiliza para definir datos geoespaciales que deben ser representados sobre la superficie de la tierra. Por otro lado, el tipo *geometry* se utiliza para representar los elementos espaciales sobre superficies planas.

A la hora de introducir datos espaciales en el sistema, SQL Server reconoce tres formatos estándar: *Well-Known Text* (WKT), *Well-Known Binary* (WKB) y *Geometry Markup Language* (GML). Para cada formato existe un conjunto de métodos, pero las funcionalidades proporcionadas son las mismas en todos los casos. En el siguiente código de ejemplo se muestra la definición de una polilínea mediante el formato WKT, y su posterior inserción en la tabla "Geoms":

```
INSERT INTO Geoms (
Geom
)
VALUES (
      geometry::STLineFromText(
            'LINESTRING(
                  53.4  -2.99,
                  53.5  -3.15,
                  53.47  -4.66
            )',
            4326
      )
)
```

Además de la creación y consulta de objetos espaciales, SQL Server dispone de numerosos métodos para analizar este tipo de datos desde diversos puntos de vista. Existe un primer grupo de métodos destinados a analizar objetos de forma individual y devolver alguna de sus propiedades. Otros métodos definen nuevos objetos mediante la combinación o modificación de objetos existentes. Es el caso de los métodos de intersección (*STIntersection()*) o unión (*STUnion()*). El último grupo de métodos está dedicado al análisis de las relaciones entre objetos. En este grupo se pueden encontrar

funciones como *STDistance()*, que calcula la distancia entre dos objetos o *STEquals()*, encargada de determinar si dos objetos son iguales.

## 3.4. REST

*Representational state transfer* (REST) (25) es un estilo de arquitectura software consistente en un conjunto de principios aplicados a componentes, conectores y datos de un sistema hipermedia distribuido como la web. Este estilo de arquitectura se aplica comúnmente al desarrollo de servicios web, como es el caso de este proyecto, y se presenta como alternativa a otras especificaciones como SOAP.

Los sistemas basados en la arquitectura REST están basados en el concepto de recurso como unidad de información. Para interactuar con los recursos utilizan los métodos de HTTP de manera explícita. La forma de operar es la siguiente: dado un recurso se usa el método POST para darlo de alta en el sistema, PUT para actualizarlo, GET para obtenerlo del sistema y DELETE para eliminarlo.

Los sistemas REST no mantienen el estado entre llamadas. Esto permite aumentar la escalabilidad y es especialmente importante en sistemas con servidores replicados donde el hecho de no almacenar el estado simplifica en gran medida el análisis y diseño de la aplicación, además de mejorar el rendimiento.

Cada recurso se identifica mediante una URI (*Uniform Resource Identifier*). La sintaxis de estas direcciones suele ser bastante intuitiva y auto-descriptiva. Generalmente reflejan la estructura jerárquica de los recursos como en un árbol de directorios. Por ejemplo, la siguiente URI identifica el recurso correspondiente a la estación meteorológica nº10 del presente proyecto:

```
http://FarmCropServer/api/estaciones/estacion/10/
```

Los servicios web basados en REST resultan más fáciles de utilizar que los basados en SOAP. Una petición SOAP es mucho más pesada que una URI de un servicio REST, pues debe ir acompañada de la descripción del servicio (WSDL) y de las cabeceras de intercambio de información. Por este motivo, generalmente los servicios SOAP se construyen mediante herramientas de desarrollo que ocultan la complejidad del protocolo. Las URIs de los servicios REST, en cambio, suelen tener una sintaxis fácilmente legible por humanos. Por otro lado, SOAP proporciona un tipado fuerte de los datos y obtiene un mejor rendimiento cuando se deben adjuntar grandes cantidades de datos en las peticiones. En resumen, REST parece ser la mejor solución

para una gran parte de los servicios web, pero algunos servicios, por sus especiales características, seguirán funcionando mejor con SOAP.

# 4.Análisis y diseño

A lo largo de este capítulo se describen las principales tareas de análisis y diseño que se han llevado a cabo durante el desarrollo del proyecto. En el primer apartado se explica la arquitectura inicial del sistema y se propone otra para futuros desarrollos. El capítulo sigue con la definición de los diagramas de clases y de casos de uso, y finaliza con una descripción de la interfaz de usuario y el modelo de navegación de la aplicación cliente FarmCrop.

## 4.1. Arquitectura del sistema

Como solución al problema planteado, se ha diseñado una arquitectura que sigue una filosofía orientada a servicios. Se pretende que este sistema sea capaz de ofrecer las diferentes funcionalidades requeridas en un momento dado por las aplicaciones cliente, a través de la correspondiente capa de servicios web.

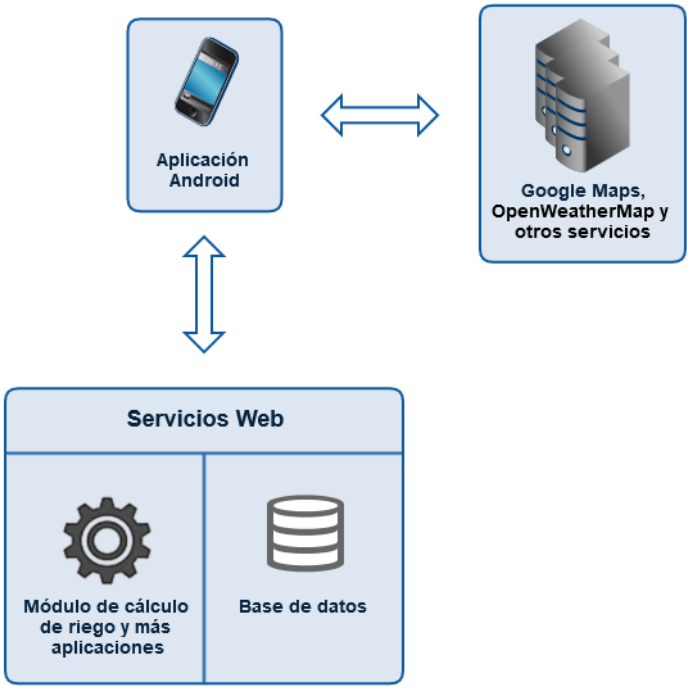



**Ilustración 15: Arquitectura del sistema I**

La arquitectura ha sido diseñada para soportar cambios y nuevos desarrollos en el sistema. Los diferentes componentes están poco acoplados entre sí, facilitando de esta forma, la reusabilidad, el mantenimiento y la ampliación de las funcionalidades existentes. Además, cada una de estos componentes puede ser desarrollado con la tecnología que resulte más conveniente sin que esto afecte a la comunicación con el resto del sistema.

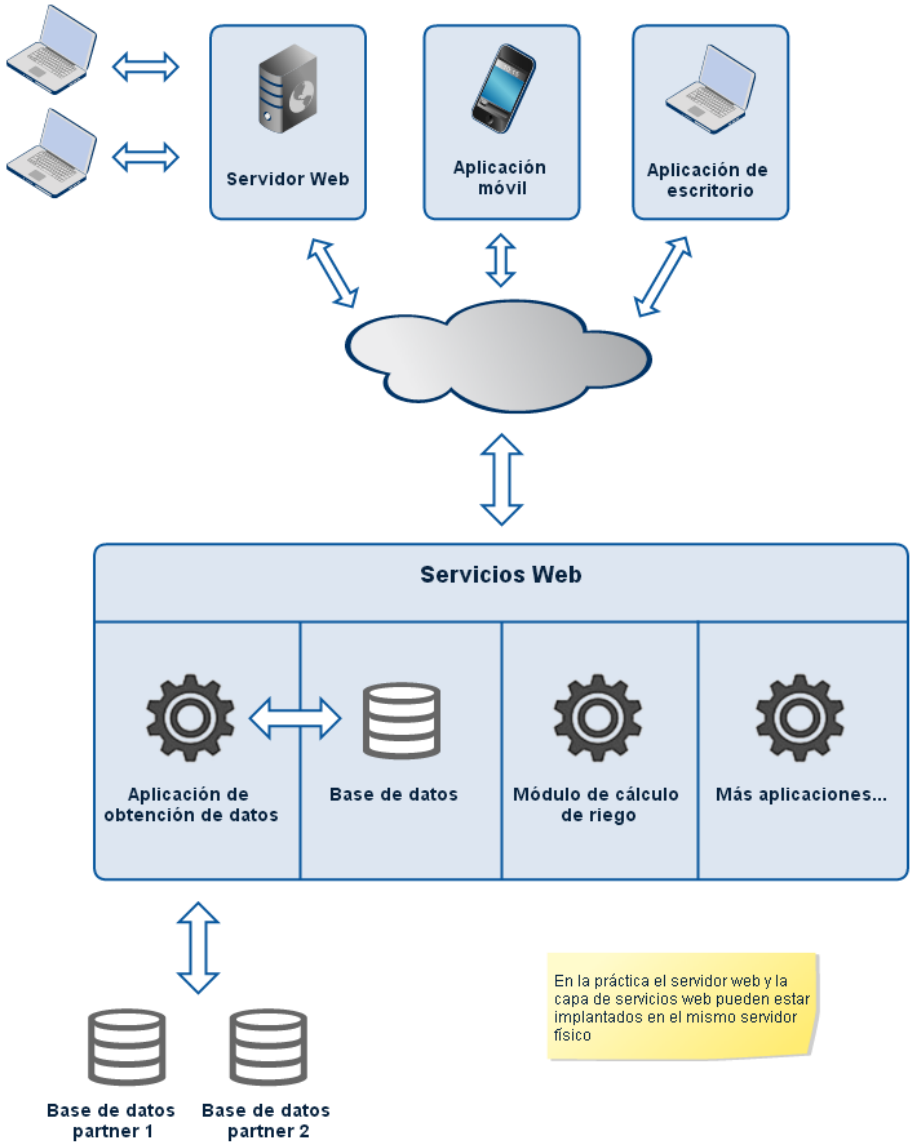



**Ilustración 16: Arquitectura del sistema II**

Como se puede apreciar en la Ilustración 15, el servidor del sistema encapsula el acceso a la base de datos y a otras aplicaciones a través de una capa de servicios web. La aplicación Android interactúa con el servidor exclusivamente a través de esta

capa de servicios. Se puede observar también que la aplicación cliente hace uso de otros servicios externos, como OpenWeatherMap (26) o Google Maps (27).

En futuros desarrollos, el sistema podría fácilmente evolucionar hacia una arquitectura como la que se muestra en la Ilustración 16. En este esquema se observa como aplicaciones cliente, de distinta naturaleza, acceden a los datos del servidor a través de diferentes soportes, pero utilizando la misma capa de servicios. Por su parte, el servidor cuenta con nuevos orígenes de datos y encapsula nuevas aplicaciones de las que puede exponer sus funcionalidades según se estime oportuno.

Se espera que la arquitectura descrita anteriormente sirva de modelo para los nuevos desarrollos que deseen integrarse en el sistema. En el caso de que la plataforma creciese con la incorporación de nuevas herramientas y funcionalidades, podría ser conveniente la creación de un ESB (*Enterprise Service Bus*) que actuara de intermediario entre las aplicaciones cliente, la base de datos y las herramientas expuestas mediante servicios web.

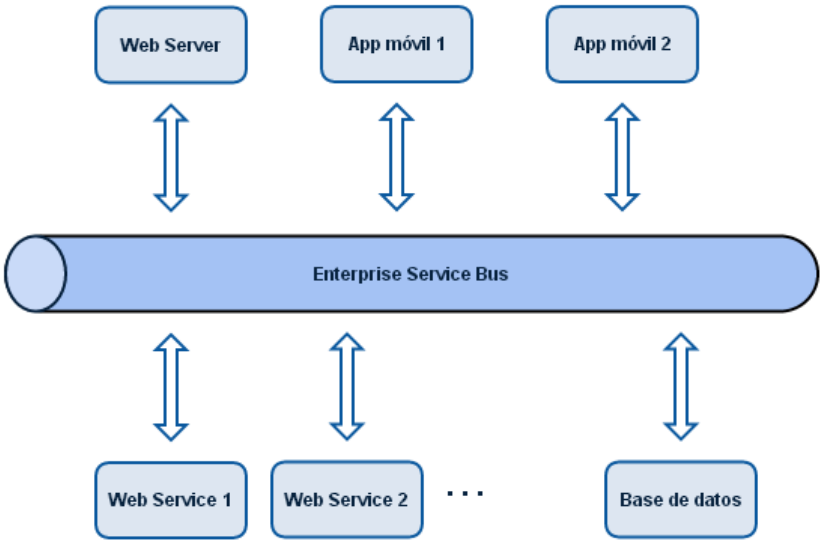



**Ilustración 17:** ***Enterprise Service Bus***

## 4.2. Casos de uso

Durante la fase de análisis, se utilizaron casos de uso para facilitar la captura de las funcionalidades y actores implicados en el sistema, así como otros requisitos funcionales de relevancia. A su vez, se desarrollaron los diagramas de casos de uso correspondientes, que si bien no ofrecen un nivel de detalle tan amplio como un caso

de uso concreto, sí que proporcionan una visión global simplificada de la interacción entre actores y funcionalidades en el sistema.

La Ilustración 18 corresponde al diagrama de casos de uso de la aplicación cliente FarmCrop. En él se puede observar la identificación de cuatro actores que interactúan con el sistema, dos de ellos (Usuario, Servidor IIAMA) con especial relevancia por el número de casos de uso en el que participan. Se trata, por una parte, del usuario que accede a la aplicación desde su dispositivo móvil y por otra, del servidor del IIAMA que se comunica con la aplicación cliente a través de servicios web.

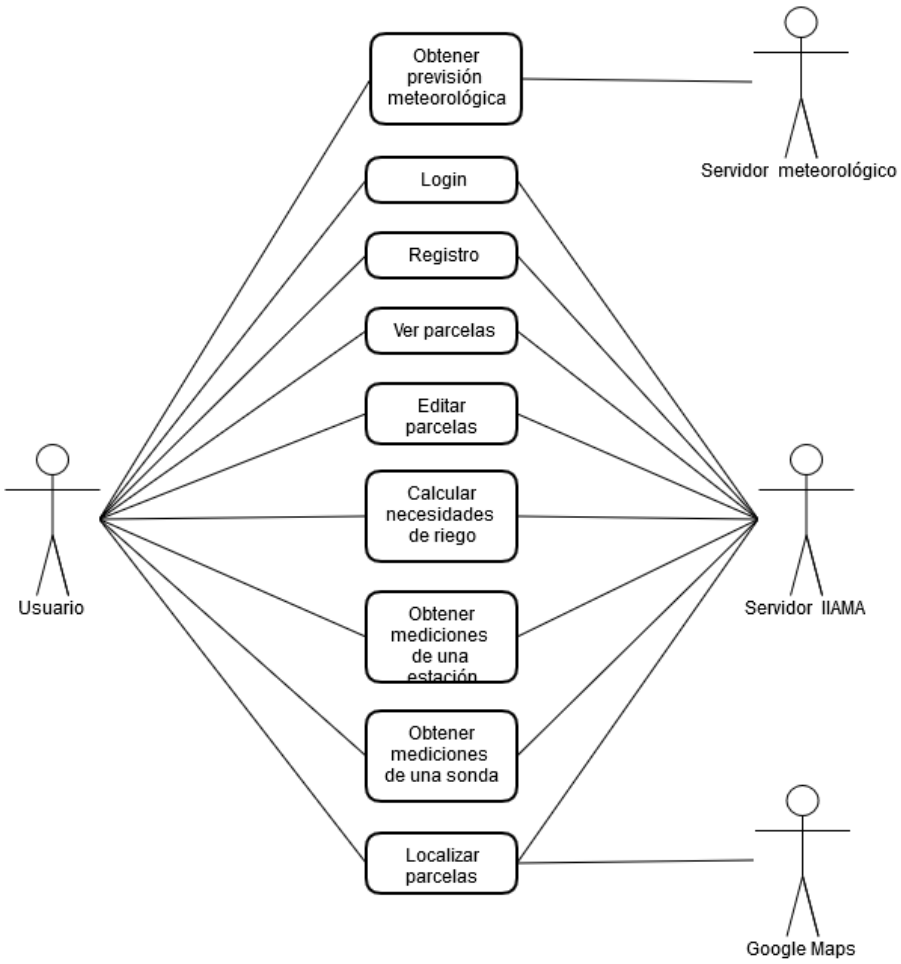



**Ilustración 18: Diagrama de casos de uso**

Además, aparecen representados otros actores en funcionalidades más concretas. Es el caso del servidor de datos meteorológicos, que proporciona datos sobre las predicciones del tiempo en la ubicación de las parcelas, o Google Maps,

servicio del que hace uso FarmCrop para representar los mapas de parcelas, estaciones y sondas.

## 4.3. Diagrama de clases

Mediante la utilización de diagramas de clases de UML es posible representar, tanto la estructura de un sistema como las relaciones existentes entre las diversas entidades del dominio implicadas. En estos diagramas se suelen representar diversos elementos de la programación orientada a objetos, como son: clases, operaciones, atributos y relaciones estáticas entre objetos. Por simplificación, el diagrama mostrado en este apartado consta exclusivamente de clases y relaciones entre objetos.

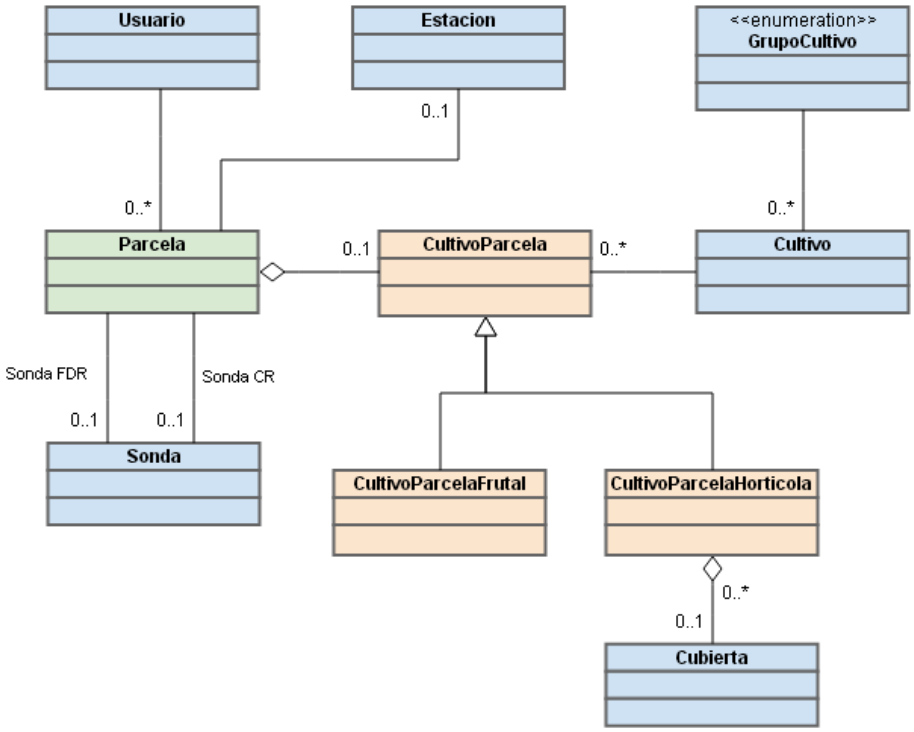



**Ilustración 19: Diagrama de clases**

En el diagrama de clases de la Ilustración 19 están representadas las principales entidades del sistema relacionadas con el modelo de datos del sector agroalimentario. Por lo tanto, no se incluyen en este diagrama otras clases utilizadas en el proyecto, que dependen de una implementación concreta y no de las relaciones inherentes al problema que se desea resolver.

Como se puede observar en el diagrama, cada usuario puede tener asociadas diversas parcelas. Cada parcela contiene, además de las relaciones mostradas en el

diagrama, un identificador, un nombre y un conjunto de coordenadas que pueden representar un punto o un polígono en función de como el usuario haya localizado la parcela. Para la primera versión de la aplicación cliente FarmCrop, se ha considerado limitar el número de estaciones meteorológicas a una por parcela. En futuras versiones la aplicación deberá proporcionar al usuario la posibilidad de asociar múltiples estaciones a una misma parcela y los cálculos de las necesidades hídricas para esa parcela deberán adaptarse en consecuencia.

En el caso de las sondas FDR (sondas de humedad del suelo) y CR (sondas de neutrones), se han añadido a la aplicación de forma experimental y no estarán disponibles en la primera versión publicada. En futuras versiones, se espera que usuarios con un perfil avanzado puedan conectar sus propias sondas a la aplicación y asociarlas a las parcelas en las que están situadas. Los datos proporcionados por estas sondas permitirán obtener las necesidades hídricas por otros medios (apartado 2.1.1) y por tanto, se podrán realizar comparaciones con las necesidades hídricas obtenidas mediante parámetros climáticos (apartado 2.1.3).

Cada parcela puede contener como máximo un objeto de tipo "*CultivoParcela*". Esta clase representa la información asociada a la plantación de un determinado cultivo en una determinada parcela. Gran parte de los parámetros de la plantación difieren según se trate de un cultivo frutal o un cultivo hortícola, por este motivo se han creado las especializaciones "*CultivoParcelaFrutal*" y "*CultivoParcelaHortícola*". Tal como se comenta en el apartado 2.1.3, el proceso de cálculo de las necesidades hídricas es diferente para cultivos frutales y cultivos hortícolas. Por el momento, la aplicación cliente solo soporta la inserción de parcelas con cultivos frutales, pues el cálculo de riego para cultivos hortícolas aún no está implementado. En el futuro ambos tipos de cultivos serán soportados.

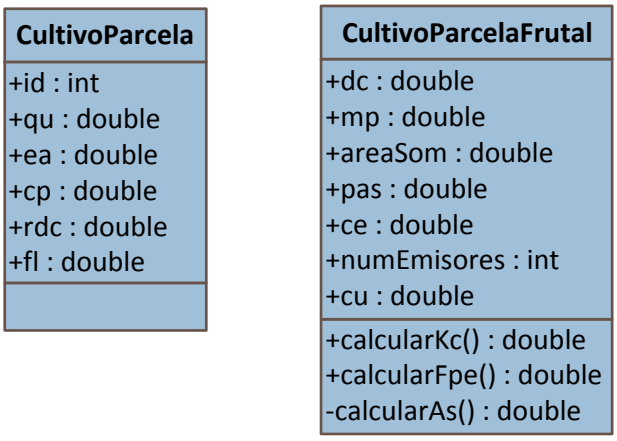



**Ilustración 20: Clases *CultivoParcela* y *CultivoParcelaFrutal***

La clase *Cultivo* representa los tipos de cultivos genéricos contemplados por el sistema. Esta clase contiene además algunos parámetros relativos a cada tipo de cultivo en cuestión. Es el caso del coeficiente de cultivo medio (kcm), la salinidad máxima admitida por el cultivo (cex), o su factor de precipitación efectiva (fpe).

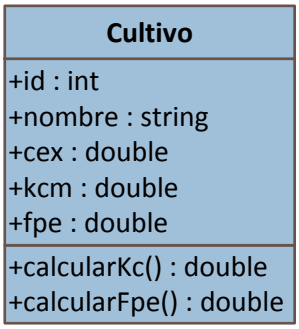



**Ilustración 21: Clase *Cultivo***

## 4.4. Mapa de navegación

En el diseño de aplicaciones para dispositivos móviles es de suma importancia la creación de un modelo que maximice la velocidad de navegación y minimice el número de pulsaciones necesarias para acceder a los datos. Todo esto sin que la Interfaz deje de ser intuitiva y consistente. De este modo se consigue mejorar en gran medida la experiencia del usuario al interactuar con la aplicación. El modelo de navegación se convierte así en un punto clave de la fase de diseño.

En la Ilustración 22 se encuentra reflejado el mapa de navegación de la aplicación cliente. En este diseño se han aplicado diversos patrones de navegación que propone Android en su documentación (28). Es el caso de la navegación horizontal (a través de gestos de arrastre en la pantalla) entre ventanas con contenido similar y mismo nivel jerárquico, o del panel de navegación principal, implementado mediante un *Navigation Drawer*.

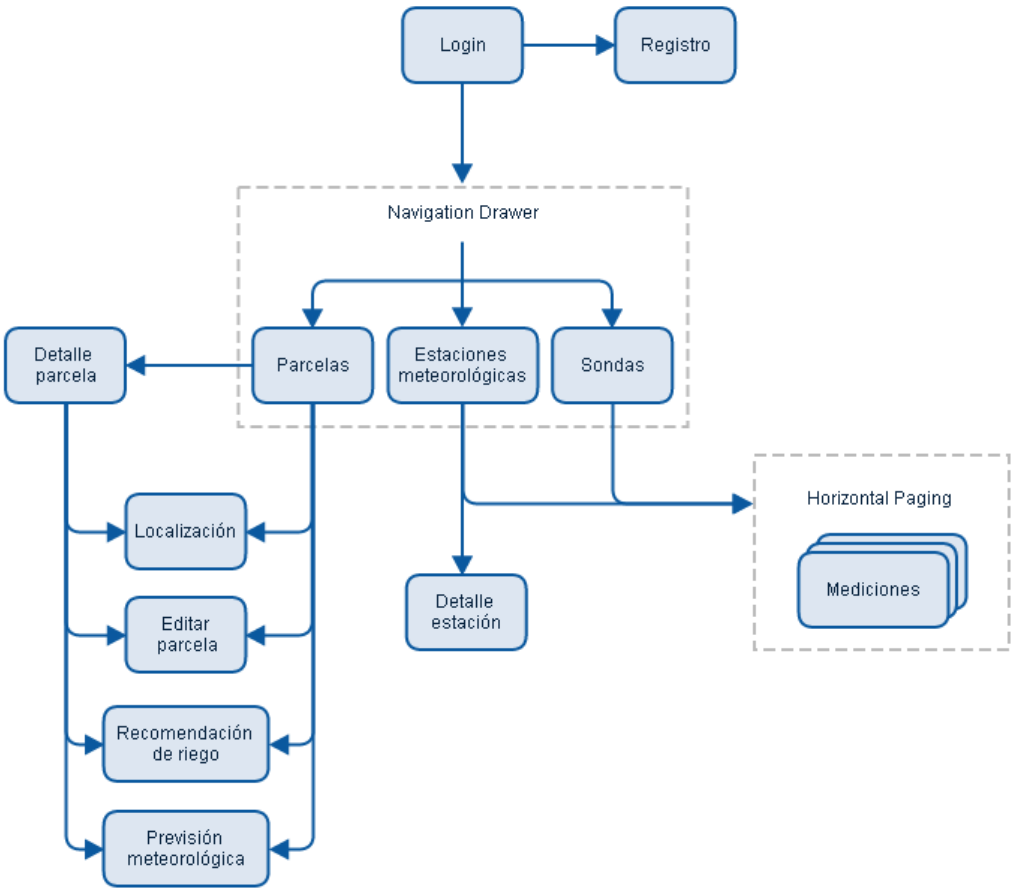



**Ilustración 22: Mapa de navegación de la aplicación**

Para conseguir un modelo de navegación consistente entre las diversas vistas principales de la aplicación, se ha optado por utilizar el patrón de navegación proporcionado por el componente *Navigation Drawer*. Este elemento forma parte de la lista de *widgets* que proporciona Android por defecto. Consiste en un panel lateral que se despliega normalmente desde el icono situado en la esquina superior izquierda de la *Action Bar*, o bien, realizando un gesto de arrastre desde el límite izquierdo de la pantalla hacia el centro de la misma.

El panel del *Navigation Drawer* proporciona una forma rápida y sencilla de navegar entre las ventanas principales de la aplicación. Es útil cuando existen varias vistas principales con un mismo nivel jerárquico. El componente se implementa en cada una de estas vistas y permite permutar rápidamente entre cada una de ellas. Como se trata de un patrón ampliamente utilizado en otras aplicaciones Android (se puede observar, por ejemplo, en Google Play), resulta familiar e intuitivo para el usuario. El comportamiento habitual de este componente puede observarse en la Ilustración 23.

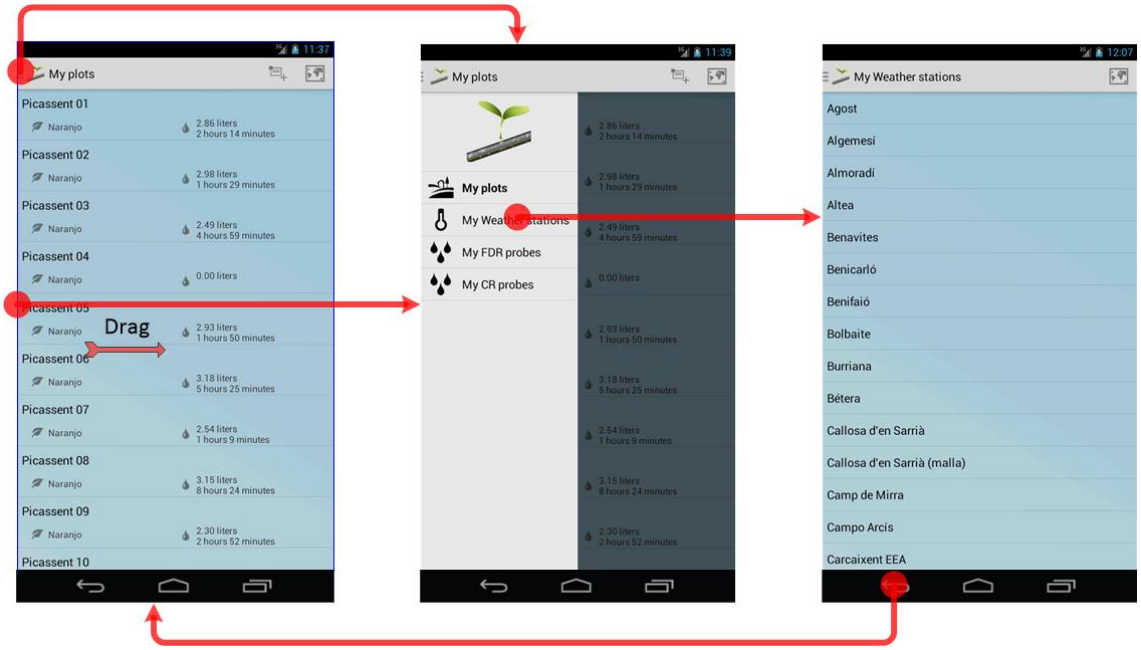



**Ilustración 23: Patrón de navegación del *Navigation Drawer***

Una vez que el usuario se ha identificado en la aplicación, accede a la vista donde aparecen listadas todas sus parcelas. Desde aquí puede navegar a alguna de las vistas detalle relacionadas con las parcelas, o bien puede desplegar el panel del *Navigation Drawer* y navegar directamente a otra vista principal. Para la versión actual de la aplicación, las vistas principales son:

- Parcelas del usuario.
- Estaciones conectadas con el sistema.
- Sondas FDR de humedad del suelo.
- Sondas CR de neutrones.

Es posible que en futuros desarrollos se creen más vistas principales para mostrar nuevos elementos relevantes del sistema. En este escenario, el componente *Navigation Drawer* es más conveniente frente a otras estructuras de navegación, pues el usuario puede ver rápidamente las principales vistas de la aplicación, y navegar entre ellas a conveniencia.

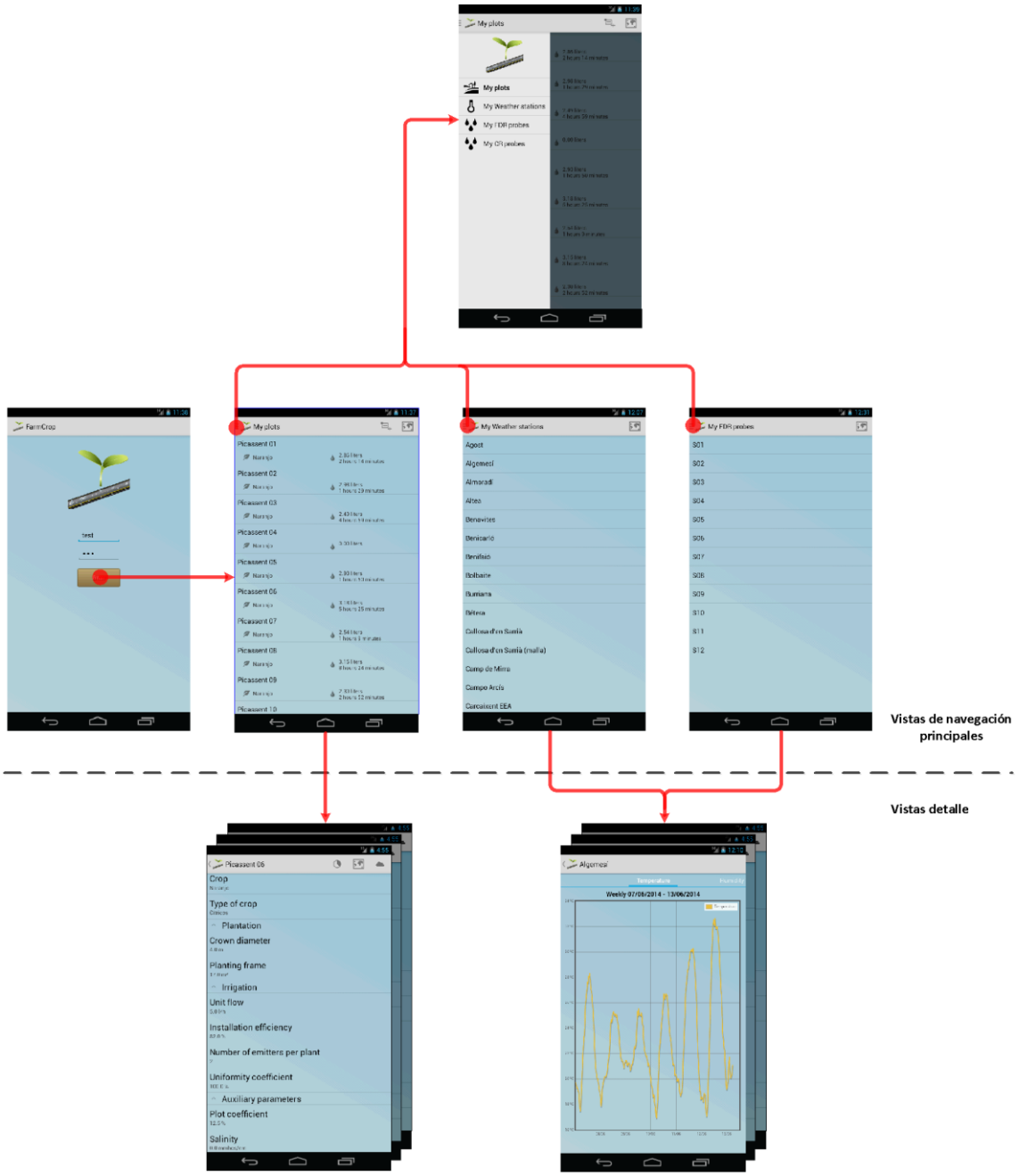



**Ilustración 24: Modelo de navegación de FarmCrop**

## 4.5. Interfaz de usuario

Durante la fase de diseño de la aplicación Android, se ha prestado una especial atención al desarrollo de las interfaces de usuario, teniendo en cuenta la importancia que tienen a la hora de crear entornos de interacción amigables, sencillos e intuitivos. El objetivo es facilitar la interacción del sistema con el usuario y que este perciba la aplicación como una herramienta útil y robusta, a la par que amigable. Si se tiene en cuenta, además, el reducido tamaño de las pantallas de los dispositivos móviles, el

diseño de una interfaz adecuada se vuelve un punto crucial en el desarrollo de la aplicación, y constituye un elemento perfectamente capaz de condicionar el éxito o el fracaso de la misma.

Partiendo de esta premisa, es importante evitar la creación de interfaces excesivamente sobrecargadas o con muchos niveles de profundidad. En su lugar, las interfaces deben resultar tan simples como lo permita la funcionalidad a desarrollar y en la medida de lo posible, deben incorporar componentes y patrones de diseño que sean comunes en otras aplicaciones. Esto permite al usuario reconocer elementos y comportamientos similares, lo que contribuye a reducir la curva de aprendizaje de la aplicación y disminuye el posible estrés causado al interactuar con un nuevo sistema.

Otro punto importante a la hora de diseñar interfaces de usuario es la aplicación de estilos. Estos deben conseguir que la aplicación sea atractiva para el usuario pero evitando en lo posible la sobrecarga de elementos decorativos que acaben por afectar negativamente a la usabilidad de la aplicación.

Con el objetivo de diseñar unas interfaces que se adecuasen a los términos mencionados anteriormente, se han realizado una serie de bocetos donde se ha intentado reflejar como debe ser la interacción del usuario con el sistema. En estos bocetos, han quedado resaltados aquellos elementos gráficos considerados de importancia a la hora de implementar las interfaces gráficas definitivas. En la Ilustración 25 se muestra uno de estos bocetos. Se puede consultar la lista completa de bocetos en el anexo I.

El proceso completo de la creación de interfaces se inició con el esbozado a mano de las interfaces de usuario. Se crearon diversos bocetos hasta que se dio con un conjunto de bocetos que de buen grado satisfacía los requisitos mencionados. En este punto los bocetos se pasaron a formato electrónico con la aplicación Balsamiq (29), especializada en la creación de esbozos de interfaces. Como paso final, las interfaces se implementaron con el editor de *layouts* del Android Developers Tools (ADT) (30) tal como se explica en el apartado 5.2.3.1.


**Ilustración 25: Ejemplo de boceto de interfaz de usuario**

# 5. Implementación

En este capítulo se describen, destacando aquellas partes más importantes, los detalles de implementación del sistema. El capítulo está separado en dos grandes bloques. En primer lugar se habla de la implementación de los diferentes componentes del servidor, para posteriormente, describir los principales elementos implementados en la aplicación cliente FarmCrop. Para el desarrollo de esta fase, siempre que ha sido posible, se han elegido tecnologías con un cierto grado de madurez, a la par que estables.

## 5.1. Implementación de los componentes del servidor

La capa de servicios del sistema ha sido desarrollada mediante el framework Web API (31) de ASP.NET MVC 4 (32). Este framework nos permite desarrollar sistemas RESTful sobre la plataforma .NET, algo que tradicionalmente ha sido complejo debido al mayor interés de .NET por los servicios basados en SOAP. El desarrollo de servicios web con Web API se asemeja mucho a una web desarrollada con ASP.NET MVC siguiendo a su vez, y como su propio nombre indica, el patrón de diseño Modelo Vista Controlador.

Para la implementación de la base de datos, se ha optado por la utilización de SQL Server 2012 en su edición Express. En una fase más avanzada del proyecto, se espera poder migrar la base de datos a otra edición que proporcione mejores prestaciones de almacenamiento, concurrencia y escalabilidad.

### 5.1.1. Implementación de servicios con ASP.NET Web API

Como se ha comentado anteriormente, Web API está basado en ASP.NET MVC y sigue su mismo patrón de desarrollo, basado en modelos, vistas y controladores. Los modelos representan, en este caso, el mismo concepto que en ASP.NET MVC, también los controladores, con la salvedad de que éstos últimos ya no devuelven vistas representables de la interfaz de usuario. Es por tanto en las vistas donde el cambio es mayor. Las vistas en Web API están representadas por los datos devueltos en formatos estándar, como XML o JSON y es el propio sistema el encargado de generar los datos en este formato, el desarrollador no tiene que realizar ninguna conversión.

#### *5.1.1.1. Pasos para la creación de servicios*

En este apartado se describen los pasos necesarios para la creación de una *web API*, ilustrando con ejemplos del propio proyecto cada una de las etapas de implementación.

En primer lugar, se necesita crear un modelo que sirva como contenedor de los datos que se quieran representar. Web API realiza, de forma automática, la conversión de estos objetos a XML, JSON o el formato que se indique. De la misma forma, se realiza la conversión del cuerpo de los mensajes HTTP recibidos a objetos del modelo. Generalmente los modelos son alimentados desde una base de datos, tal como ocurre en el presente proyecto. El siguiente ejemplo pertenece al modelo de la clase *Parcela*:

```
namespace IIAMA_WebAPI.Models
{
    public class Parcela : IParseableDesdeDB
    {

        public int Id { get; set; }
        public string Nombre { get; set; }
        public int EstacionReferencia { get; set; }
        public int SondaFDR { get; set; }
        public int SondaCR { get; set; }
        public string EmailResponsableRiego { get; set; }
        public List<SpatialUtils.LatLong> Coordenadas { get; set; }
       …
    }
}
```

El siguiente paso es definir un controlador, que será el objeto del sistema encargado de gestionar las peticiones HTTP recibidas. A la hora de desarrollar este componente, es importante tener en cuenta los siguientes puntos:

- A diferencia de ASP.NET MVC, los controladores de Web API deben heredar de la clase *ApiController* en lugar de *Controller*.
- Es habitual, aunque no obligatorio, la creación de un controlador por cada clase del modelo.
- Salvo que se establezca lo contrario, cada método definido en el controlador equivaldrá a un servicio de la API.
- Existen numerosas etiquetas con las que se pueden marcar los métodos y que permiten configurar en gran medida cada servicio.

En el siguiente ejemplo se muestra la implementación de dos de los servicios del controlador *ParcelasController*. El primero de ellos responde a una petición de tipo GET (indicado por el decorador *[HttpGet]*), recibe el identificador de una parcela y devuelve la parcela completa. En el segundo método se encuentra parte de la implementación del servicio de creación de nuevas parcelas con cultivos frutales. Recibe, desde el cuerpo de una petición POST, un objeto compuesto formado por una parcela y un cultivo. Junto con el identificador del usuario, se procede a su inserción en la base de datos y se devuelve un mensaje HTTP indicando el éxito o no de la operación:

```
[HttpGet]
public Parcela ObtenerParcela(int id)
{
    return this.parcelasProvider.obtenerParcela(id);
}


[HttpPost]
public HttpResponseMessage NuevaParcelaCultivoFrutal(
            HttpRequestMessage request,
            [FromBody] ParcelaCultivoFrutalRequest dataRequest)
{
    …

    resultado = this.parcelasProvider.nuevaParcelaCultivoFrutal(
                        dataRequest.Parcela,
                        dataRequest.Cultivo,
                        usuarioAutentificado.Id);

    return resultado ?
           request.CreateResponse(HttpStatusCode.OK) :
           request.CreateResponse(HttpStatusCode.InternalServerError);
}
```

Tras estos pasos ya se podría disponer de una web API publicada en las rutas definidas por defecto en el proyecto. En el siguiente apartado se trata con más detalle la configuración de las rutas de los servicios. En el caso del servicio *ObtenerParcela()*, la ruta que tiene asignada es la siguiente:

```
http://FarmCropServer/api/Parcelas/ObtenerParcela/{id}
```

Tal como se ha indicado anteriormente, tras la ejecución del servicio, el sistema se encarga automáticamente de transformar el objeto *Parcela* devuelto por el método, al formato de datos especificado en la cabecera correspondiente del mensaje HTTP. Por ejemplo, el siguiente fragmento corresponde a la respuesta dada por el servidor

tras la invocación del servicio anterior y representa la serialización a XML de la parcela con identificador 3034:

```
<Parcela>
	<Coordenadas>…</Coordenadas>
	<EmailResponsableRiego>sergio.kma@gmail.com</EmailResponsableRiego>
	<EstacionReferencia>54</EstacionReferencia>
	<Id>3034</Id>
	<Nombre>Picassent 01</Nombre>
	<SondaCR>1</SondaCR>
	<SondaFDR>1</SondaFDR>
</Parcela>
```

#### *5.1.1.2. Enrutamiento de peticiones HTTP*

Cuando el framework Web API recibe una petición HTTP, esta se redirecciona a un servicio en concreto, siempre y cuando la petición esté bien formulada. Para determinar qué servicio debe ser invocado, el sistema cuenta con una tabla de enrutamiento donde están definidas una serie de reglas y condiciones. Esta configuración puede encontrarse en el archivo *"WebApiConfig.cs"* del proyecto.

El siguiente fragmento de código corresponde a la tabla de enrutamiento seguida en este proyecto. Para encaminar los servicios desarrollados en el sistema ha sido suficiente con un solo patrón, pero, en caso de ser necesario, pueden definirse tantos como se crea conveniente. Las expresiones *{controller}* y *{action}* son variables que se sustituyen por los nombres de los controladores y de los servicios respectivamente. La variable *{id}* es un parámetro que reciben los servicios directamente desde la URL y está calificado como opcional, pues no todos los servicios hacen uso de este parámetro.

```
	config.Routes.MapHttpRoute(
		name: "ActionApi",
		routeTemplate: "api/{controller}/{action}/{id}",
		defaults: new { id = RouteParameter.Optional }
	);
```

#### *5.1.1.3. Autentificación*

Los servicios desarrollados han sido protegidos mediante un mecanismo de autentificación que permite validar al usuario que realiza la petición, y dado el caso, puede restringir el acceso a unos determinados recursos en función de los roles o permisos que posea el usuario. Existen diversas opciones a la hora de proporcionar seguridad a los servicios desarrollados con Web API. En este proyecto se ha seguido la *Basic Authentication* de HTTP (33), por ofrecer la suficiente funcionalidad y resultar adecuado a la magnitud del proyecto. Otros mecanismos de autentificación interesantes son OpenID (34) u OAuth (35).

Los servicios con una filosofía RESTful carecen de estado, lo que implica que cada llamada realizada se trata de forma aislada y no se almacena ninguna información referente al emisor. Por este motivo es necesario que cada petición HTTP efectuada lleve consigo los credenciales del usuario, como nombres de usuario, contraseñas, certificados o *tokens*, dependiendo del mecanismo de autentificación utilizado.

La autentificación con *Basic HTTP Authentication* es relativamente sencilla. Consiste en la introducción de una nueva cabecera en la petición por parte del emisor, donde se indique el nombre de usuario y la contraseña. Los pasos concretos que se deben seguir son los siguientes:

- Crear una cadena de caracteres con nombre de usuario y contraseña de la siguiente forma: “nombreUsuario:contraseña”.
- Codificar la cadena de caracteres en Base 64 y añadir la palabra “*Basic*” al inicio.
- Incluir en la petición HTTP una nueva cabecera de autorización con la cadena codificada.

La cabecera resultante tendrá un aspecto similar al siguiente ejemplo:

```
Authorization: Basic anNtaXRoOlBvcGNvcm4=
```

Existen diversas formas de tratar las cabeceras de certificación en el framework Web API. Por una parte, se puede gestionar la autentificación a nivel de controlador o servicio. Con esta aproximación se consigue una buena granularidad a la hora de gestionar los permisos y roles del usuario, pero obliga a mezclar el código de autentificación con la lógica del servicio, resultando en una implementación más desordenada.

Otra opción consiste en la implementación de un manejador de mensajes, que filtre todas las peticiones HTTP, para comprobar las credenciales antes de que los mensajes lleguen a los servicios. Con esta aproximación, de mayor granularidad que la anterior, se consigue tener todo el código de autentificación centralizado y ordenado. En el desarrollo del proyecto se ha utilizado esta segunda opción, por considerarse suficiente para los requisitos de seguridad de los servicios implementados, si bien ambas soluciones pueden coexistir sin problemas.

Para la implementación de un manejador de mensajes en Web API es necesario crear una subclase que herede de la clase *DelegateHandler*. Como se ha indicado anteriormente, este objeto será el encargado de interceptar las peticiones HTTP y comprobar que los credenciales sean correctos. El manejador también será el encargado de responder a las peticiones que carezcan de cabecera de autentificación, indicando la necesidad de la misma mediante la inclusión de la cabecera "*WWW-Authenticate*" en la respuesta HTTP.

La subclase generada debe sobrescribir el método *SendAsync()*, donde se debe comprobar la existencia de la cabecera de autentificación y posteriormente, acreditar que el nombre de usuario y la contraseña contenidos en la cabecera son correctos. Si cualquiera de las dos comprobaciones resulta insatisfactoria, *SendAsync()* responderá con el mensaje indicado en el párrafo anterior. En el siguiente ejemplo se muestra la implementación de este método en el manejador de autentificación del proyecto:

```
protected override Task<HttpResponseMessage> SendAsync(
             HttpRequestMessage request,
             System.Threading.CancellationToken cancellationToken)
{
       if (request.Headers.Authorization != null &&
           request.Headers.Authorization.Scheme == "Basic")
      {
           var credentials = ParseCredentials(request.Headers.Authorization);

           if (Authorize(credentials.Username, credentials.Password))
           {
               return base.SendAsync(request, cancellationToken);
           }
       }

      return Task<HttpResponseMessage>.Factory.StartNew(
           () =>
           {
               var response =
                      new HttpResponseMessage(HttpStatusCode.Unauthorized);

               response.Headers.Add(
                      "WWW-Authenticate",
                      string.Format("Basic realm=\"{0}\"", Realm));

               return response;
```

```
            }
        );
}
```

El método *Authorize()* del ejemplo anterior es un método abstracto. Esto permite implementar diversos procesos de autentificación, manteniendo el resto de la lógica intacta a través de la especialización de la clase. Una vez implementado el manejador, éste debe ubicarse en el archivo global de configuración junto con la tabla de enrutamiento y otros elementos del sistema:

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();

        WebApiConfig.Register(GlobalConfiguration.Configuration);
        GlobalConfiguration.Configuration.MessageHandlers.Add(
             new IIAMABasicAuthHandler() );
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);

        …
    }
}
```

### 5.1.2. Módulo de acceso a datos

La conexión entre la aplicación del servidor y la base de datos se realiza a través de un módulo de acceso a datos, implementado con funciones genéricas, que es capaz de gestionar automáticamente la transformación entre registros de la base de datos y objetos del modelo de datos del sistema. Este módulo proporciona la funcionalidad y el rendimiento adecuados para el tamaño actual del proyecto. Si el sistema creciese con futuros desarrollos, podría plantearse la implantación de algún sistema de mapeo objeto-relacional (ORM), como Nhibernate (36) o Entity Framework (37), que gestionase el acceso a la base de datos.

Gran parte del funcionamiento del módulo de acceso a datos está basado en la implementación de dos interfaces. En primer lugar, toda entidad del sistema que necesite ser almacenada en la base de datos debe implementar la interfaz *IParseableDesdeDB*:

```
public interface IParseableDesdeDB
{
    void obtenerDesdeDataReader(IDataRecord rdr);
}
```

Como se puede observar, esta interfaz posee un solo método, que recibe como argumento un cursor genérico con los registros leídos desde la base de datos. El hecho de que el cursor sea genérico es importante, pues desvincula a la clase de la utilización de una base de datos concreta. En la implementación de este método, la clase debe alimentar sus campos con los datos obtenidos desde el cursor. En el siguiente ejemplo se muestra un fragmento de la implementación del método *obtenerDesdeDataReader* en la clase *Parcela*:

```
public void obtenerDesdeDataReader(IDataRecord rdr)
{
    this.Id = Convert.ToInt32(rdr[0]);
    this.Nombre = Convert.ToString(rdr[1]);
    this.EstacionReferencia = rdr[2] != DBNull.Value ?
         Convert.ToInt32(rdr[2]) : -1;
    this.SondaFDR = rdr[3] != DBNull.Value ?
         Convert.ToInt32(rdr[3]) : -1;
    this.SondaCR = rdr[4] != DBNull.Value ? Convert.ToInt32(rdr[4]) : -1;
    …
}
```

La segunda interfaz que debe ser implementada es *IDataProvider*. Esta interfaz define las operaciones básicas de acceso a datos que contempla el sistema. Permite abstraer el conjunto de operaciones a realizar de su implementación concreta, que dependerá del sistema de gestión de bases de datos empleado. Por tanto, se debe realizar una implementación de esta interfaz por cada tipo de base de datos que utilice el sistema:

```
public interface IDataProvider
{
    List<T> obtenerListaEntidades<T>(string query, Dictionary<string,
        Object> parametros) where T : IParseableDesdeDB, new();

    T obtenerEntidad<T>(string query, Dictionary<string, Object>
        parametros) where T : IParseableDesdeDB, new();

    bool ejecutarComando(string query, Dictionary<string, Object>
        parametros);
```

```
        bool ejecutarProcAlmacenado(string proc, Dictionary<string, Object>
            parametros);

        T ejecutarEscalar<T>(string query, Dictionary<string, Object>
            parametros);
    }
```

Las operaciones de la interfaz que devuelven objetos del modelo de datos (*obtenerEntidad* y *obtenerListaEntidades*) son funciones genéricas, pero tienen definida la condición de que el tipo de datos con el que se invoque la función debe implementar, obligatoriamente, la interfaz *IParseableDesdeDB* a la que se ha hecho referencia anteriormente. La palabra clave *new()* en la definición de las operaciones significa que éstas tienen la facultad de crear nuevos objetos del tipo de datos pasado a la operación, siempre y cuando el tipo de datos cuente con un constructor sin parámetros. En este escenario, una implementación concreta de la interfaz *IDataProvider* tiene la capacidad de generar nuevos objetos del modelo a partir de los datos obtenidos desde la base de datos.

A continuación se muestra la invocación de una de estas operaciones. En este caso, la función ejecutada devolvería un objeto de tipo *Parcela*, tras la especificación de una determinada consulta SQL y de los parámetros de la consulta en forma de diccionario:

```
    provider.obtenerEntidad<Parcela>(SQL_QUERY, parametros_query);
```

La implementación concreta de la interfaz *IDataProvider* se ha realizado para el SGDB que utiliza actualmente el sistema: Microsoft SQL Server. Esta clase implementa todos los métodos de la interfaz y tiene en cuenta los detalles de programación específicos de este SGDB en cuestión. Además, para cada entidad del sistema se ha creado una clase que gestiona todas las operaciones propias de la entidad con la base de datos, a través del acceso genérico proporcionado por un objeto *IDataProvider*. En la Ilustración 26 puede observarse el diagrama de clases que ilustra la estructura del módulo de acceso a datos descrito en esta sección.

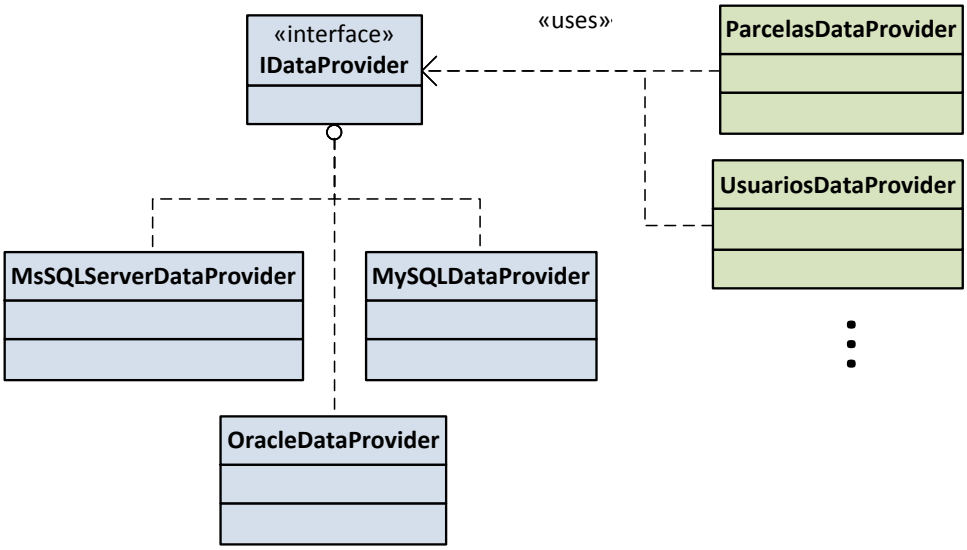



**Ilustración 26: Diagrama de clases del módulo de datos**

### 5.1.3. Base de datos

Como se ha comentado anteriormente, se ha empleado MS SQL Server 2012 como soporte para la persistencia de datos. En la implementación de las tablas de la base de datos se ha seguido, a grandes rasgos, el diagrama de clases definido en el capítulo de diseño. Para representar la herencia, se han creado claves ajenas en relaciones uno a uno, tal como se explica en el artículo *"Implementing Table Inheritance in SQL Server"* de la página sqlteam.com (38). El esquema resultante puede observarse en la Ilustración 27.

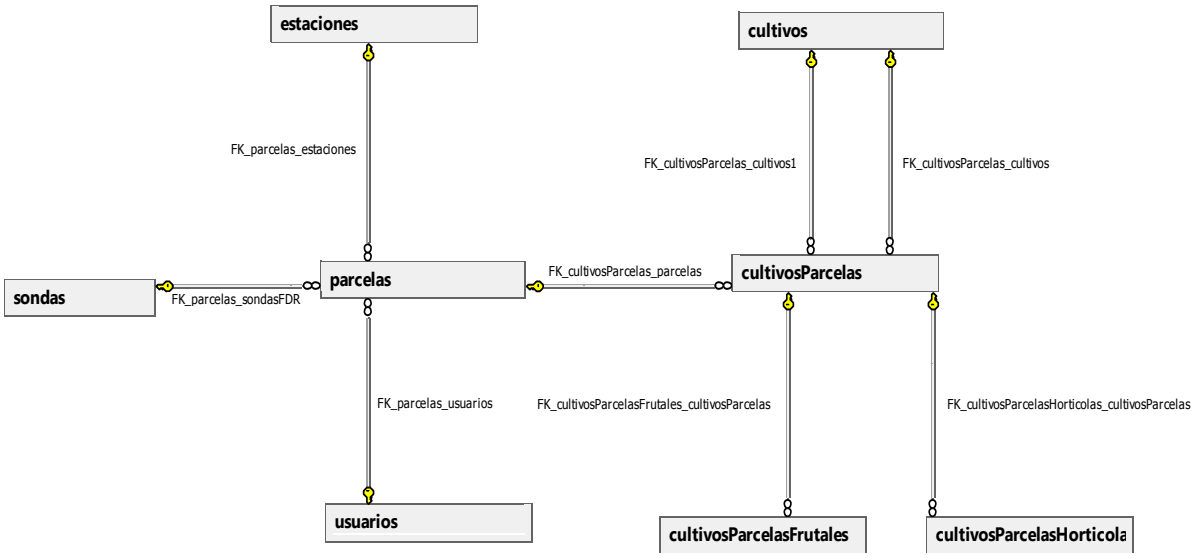



**Ilustración 27: Diagrama de la base de datos**

Las operaciones más complejas se han implementado mediante transacciones ubicadas en procedimientos almacenados. Estas operaciones son sobre todo, inserciones y modificaciones que deben ser reflejadas en varias tablas para mantener la integridad de los datos. Las operaciones de consulta, por ser generalmente más simples, se albergan en su mayoría en las clases proveedoras de datos de la capa de servicios. En un futuro podrían trasladarse todas las consultas a procedimientos almacenados, con el objetivo de clarificar el código y conseguir una mayor separación entre capas.

En el desarrollo del proyecto se han utilizado ampliamente las funciones geoespaciales, incluidas de serie en SQL Server, para almacenar la ubicación de parcelas, estaciones y sondas. En concreto, se han utilizado funciones para crear, consultar y modificar objetos espaciales, además de algunas funciones de análisis y transformación de estos objetos. Con esto se ha conseguido simplificar en gran medida el tratamiento de este tipo de datos. En el siguiente fragmento de código se muestra una consulta donde se utilizan diversas funciones geoespaciales. La consulta comprueba que los puntos de un polígono estén definidos en el orden correcto y en el caso de que no sea así, invierte su orden.

```
DECLARE @geom geography = geography::STGeomFromText(@poligono, @srid)
SELECT
    CASE
        WHEN @geom.EnvelopeAngle() >= 180 THEN
             @geom.ReorientObject().STAsText()
        ELSE @geom.STAsText()
    END
```

## 5.2. Implementación de la aplicación FarmCrop

Como ya se ha mencionado en apartados anteriores, la aplicación cliente ha sido desarrollada para la plataforma móvil Android. La aplicación es compatible con los dispositivos que utilicen este sistema operativo hasta la versión 2.2 (Froyo) y ha sido compilada con el nivel 19 de la API de Android.

### *5.2.1.* Implementación de *activities*

En este apartado se describen los pasos necesarios para la creación de *activities* (ver sección 3.1.2.1) y se comenta la estructura de las actividades seguida en este

proyecto. También se habla de la implementación de *fragments* y de la interfaz *Parcelable*, que hace posible el paso de objetos complejos entre actividades y otros componentes.

#### *5.2.1.1. Pasos para la creación de una activity*

En primer lugar, para crear una actividad se debe crear una clase que herede de *Activity* o una subclase de ésta. Posteriormente se deben implementar una serie de métodos de *callback* que el sistema llama cuando la actividad cambia entre varios de los estados de su ciclo de vida (ver Ilustración 5). No es necesario implementarlos todos, solo aquellos que el desarrollador necesite para modificar el comportamiento de la actividad o para manejar errores inesperados que puedan interrumpir la ejecución de la misma. Es especialmente importante el método *onCreate()*. Este método se llama cuando el sistema crea la actividad. En él se deben inicializar los principales componentes de la actividad y cargar la interfaz de la misma mediante el método *setContentView()* (generalmente a través de un recurso *layout*).

Otros métodos de *callback* importantes son *onPause()* y *onSaveInstanceState()*. El primero de ellos es el único método que el sistema asegura que será ejecutado en el proceso de destrucción de una actividad. Aunque que sea ejecutado no significa necesariamente que la actividad vaya a ser destruida, es un buen lugar para salvar los datos relevantes que todavía no hayan sido guardados.

El segundo método se ejecuta cuando la actividad va a ser destruida, en un contexto donde es posible que la actividad sea restaurada. Por ejemplo, cuando el usuario navega hacia delante y hacia atrás en un escenario maestro-detalle. El usuario espera que al volver a la actividad maestra, ésta se encuentre tal cual la dejó, pero es posible que durante el proceso de navegación, la actividad maestra haya sido destruida por falta de memoria. En este caso el desarrollador puede utilizar el método *onSaveInstanceState()* y escribir en el objeto *bundle* recibido como parámetro, los datos relevantes al estado de la actividad. Una vez hecho esto, si el sistema destruye la actividad y posteriormente vuelve a crearla, el objeto *bundle* con los datos del estado anterior se pasa como parámetro al método *onCreate()*, desde donde el desarrollador puede replicar la apariencia de la actividad destruida a partir de los datos del *bundle*. Este proceso es totalmente transparente para el usuario, que tiene la sensación de visualizar la misma actividad que dejó atrás.

En el siguiente fragmento de código se muestra parte de la implementación de los métodos *onCreate()* y *onSaveInstanceState()* pertenecientes a la actividad *MapaSondasActivity*. En este ejemplo se pueden apreciar algunos puntos comentados anteriormente, como la carga del *layout* de la actividad (*simple_map_layout*) por

medio del método *setContentView()*, la implementación de *onSaveInstanceState()* para guardar el estado de la actividad o la restauración del estado anterior a través del *bundle* recibido en el método *onCreate()*:

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

	if( savedInstanceState != null )
	{
		this.dictSondas =
			savedInstanceState.getSparseParcelableArray("dictSondas");
		this.verSondasCR =
			savedInstanceState.getBoolean("verSondasCR", false);
	 }

	…
}


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

	outState.putSparseParcelableArray("dictSondas", this.dictSondas);
	outState.putBoolean("verSondasCR", this.verSondasCR);
}
```

Para que una actividad sea accesible para el sistema, ésta debe ser declarada en el archivo *manifest* del proyecto. Existen numerosos atributos que pueden ser definidos para la actividad en este archivo. Algunos de los más habituales son: nombre (éste además es obligatorio), etiqueta, icono o estilo de la actividad. Por ejemplo, la definición de la actividad *ListaParcelasActivity* en el *manifest* de FarmCrop queda de la siguiente forma:

```
<activity

    android:name="org.iiama.calcregapp.activities.ListaParcelasActivity"
    android:launchMode="singleTop"
    android:label="@string/title_activity_lista_parcelas"
    android:parentActivityName=
                "org.iiama.calcregapp.activities.LoginActivity" >

    <!-- Parent activity meta-data to support API level 7+ -->
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
```

```
        android:value="org.iiama.calcregapp.activities.LoginActivity" />

</activity>
```

Como último paso, es necesario proporcionar a la actividad una interfaz de usuario que, como se ha comentado anteriormente, se puede cargar a través del método *setContentView()* de la propia actividad. La interfaz puede ser definida programáticamente a través del código, o más comúnmente, por medio de un recurso de tipo *layout*. Una vez seguidos estos pasos, puede utilizarse el método *startActivity()* para lanzar la nueva actividad desde otro componente.

#### *5.2.1.2. Jerarquía de las actividades de FarmCrop*

Las actividades del proyecto están organizadas mediante una jerarquía de clases (Ilustración 28), permitiendo así una mejor organización del código a través de la agrupación del código común en las actividades padre. De esta forma, se ha creado una actividad raíz, denominada *BaseActivity*, que contiene las funcionalidades básicas que toda actividad de la aplicación debe poseer; como las tareas comunes relacionadas con el usuario autentificado, patrones de navegación que se aplican en toda la aplicación, opciones de menú visibles en cualquier actividad, etc.

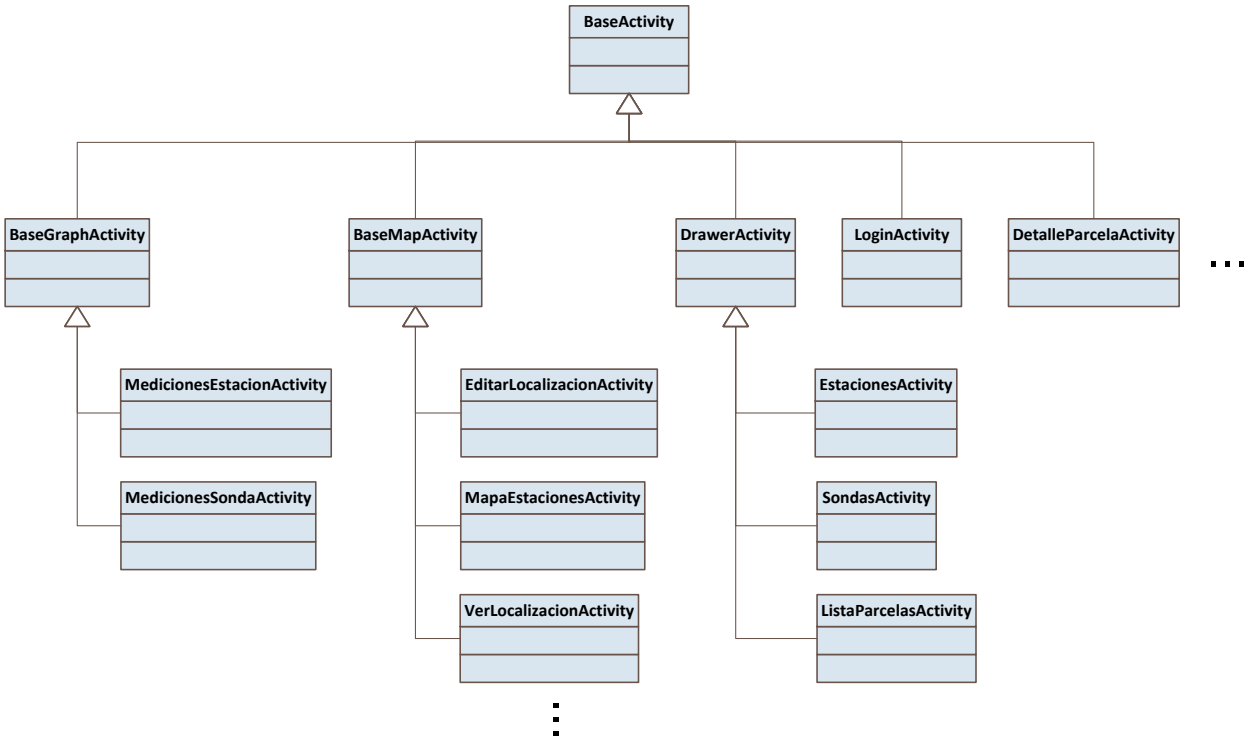



**Ilustración 28: Jerarquía de clases de las actividades**

Otras actividades de la jerarquía con funcionalidad común son:

- *DrawerActivity*: Las actividades principales de la aplicación que cuentan con el componente *Navigation Drawer*, extienden esta actividad para tener acceso a las funcionalidades que proporciona este panel de navegación.
- *BaseMapActivity*: Esta actividad reúne funciones y métodos de uso común en aquellas actividades centradas en la visualización de mapas.
- *BaseGraphActivity*: Reúne el código común referente a las actividades que trabajan con gráficos.

La actividad raíz de FarmCrop, *BaseActivity* extiende a su vez una de las clases de la API de Android: *ActionBarActivity*. Esta clase pertenece a la versión 7 de la librería de soporte de Android para versiones anteriores de la API y proporciona acceso a las características de *ActionBar* en dispositivos con un nivel de API menor que 11. A su vez, *ActionBarActivity* es una subclase de *FragmentActivity*, que pertenece a la versión 4 de la librería de soporte y proporciona las funcionalidades necesarias para poder trabajar con *Fragments* en niveles de API inferiores a 11 (ver Ilustración 29).

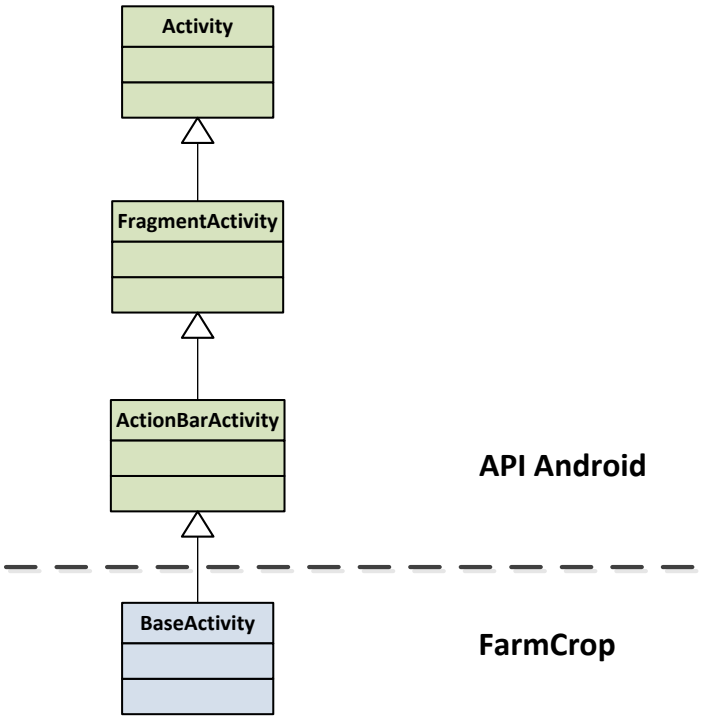



**Ilustración 29: Superclases de *BaseActivity***

#### 5.2.1.3. *Implementación de fragments*

Los *fragments* son porciones reutilizables de una actividad, tienen un funcionamiento similar a éstas y cuentan con su propio ciclo de vida y sus propios métodos de *callback* (ver apartado 3.1.2.1). En la aplicación se han utilizado para albergar los gráficos en las actividades de mediciones de estaciones y sondas. En estas actividades existe un control de desplazamiento horizontal (*ViewPager*) que mediante un gesto de arrastre por parte del usuario, cambia de una gráfica a otra sin cambiar de actividad. En cada transición, realmente, lo que se visualiza es un nuevo *fragment*. En la Ilustración 30 se ha remarcado la ubicación de los *fragments* dentro de la actividad *MedicionesEstacionActivity* durante un desplazamiento horizontal.

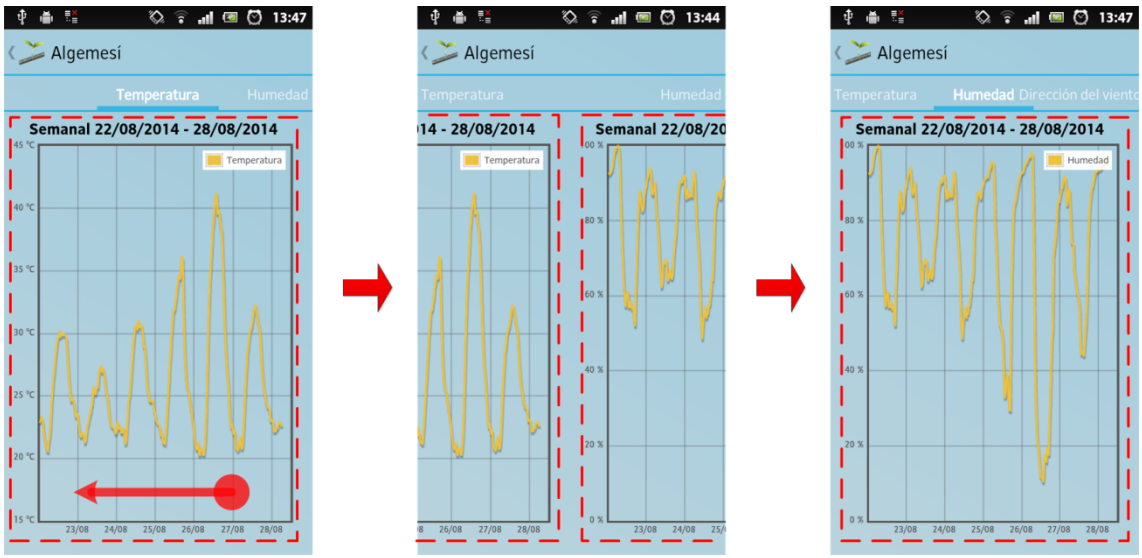



**Ilustración 30: Varios *GraphFragment* en una actividad de mediciones**

La implementación de un *fragment* es muy similar a la de una actividad. Si bien los nuevos *fragments* creados deben heredar de la clase *Fragment* en lugar de la clase *Activity*. Los métodos de *callback* también recuerdan a los de una actividad, pero además cuentan con unos cuantos métodos nuevos. Entre ellos destaca la función *onCreateView()*, que devuelve un objeto de tipo *View*. Es aquí donde el desarrollador debe cargar la interfaz propia del *fragment*. En el siguiente ejemplo se muestra la implementación de esta función en el *fragment GraphFragment*:

```
@Override
public View onCreateView(
                    LayoutInflater inflater,
                    ViewGroup container,
                    Bundle savedInstanceState)
{
    ViewGroup rootView =
```

```
        (ViewGroup) inflater.inflate(R.layout.graph_layout, container, false);

    return rootView;
}
```

#### *5.2.1.4. Interfaz Parcelable*

Como se ha comentado en el apartado 3.1.2.4, para iniciar o activar los diferentes componentes de una aplicación se utilizan objetos de tipo *intent*. Estos objetos pueden transmitir parámetros al componente de destino, siempre que éstos sean de un tipo básico (como *int, boolean, String,* etc.). Si se desea transmitir objetos complejos entre componentes, los objetos en cuestión deben implementar la interfaz de Java *Serializable*, o bien la interfaz *Parcelable* perteneciente a la API de Android.

En numerosas ocasiones durante la implementación del proyecto, ha sido necesario transferir elementos complejos (como parcelas, estaciones, usuarios, etc.) entre las diversas actividades o *fragments* de la aplicación. Por este motivo, la mayoría de objetos del modelo de datos implementa la interfaz *Parcelable*, posibilitando así este tipo de movimientos. A continuación se explica el funcionamiento de esta interfaz. Los ejemplos de implementación de la interfaz pertenecen a la clase *Estacion* del modelo de datos.

En primer lugar se deben implementar los métodos *describeContents()* y *writeToParcel()*. La primera función debe devolver un 0 en todas las implementaciones, a no ser que el objeto contenga descriptores de fichero, en cuyo caso debe retornar la constante *CONTENTS_FILE_DESCRIPTOR*. El método *writeToParcel()* obtiene como argumento de entrada un objeto de tipo *Parcel*, que es en realidad un contenedor donde se deben escribir los datos del objeto. En el siguiente fragmento de código se puede observar la implementación de ambos métodos:

```
@Override
public int describeContents()
{
        return 0;
}


@Override
public void writeToParcel(Parcel dest, int flags)
{
        dest.writeInt(this._id);
        dest.writeString(this.nombre);
        dest.writeInt(this.altitud);
```

```
            dest.writeString(this.fechaInstalacion);
            dest.writeParcelable(this.coordenadas, flags);
      }
```

A continuación se ha de declarar un objeto estático y público que implemente la interfaz *Parcelable.Creator*. Este campo va a posibilitar la creación de nuevas instancias y nuevos vectores de la clase objetivo (en este caso permitirá la creación de nuevas instancias de la clase *Estacion*). La implementación típica de este objeto es la siguiente:

```
      public static final Parcelable.Creator<Estacion> CREATOR =
            new Parcelable.Creator<Estacion>() {

        public Estacion createFromParcel(Parcel in)
        {
            return new Estacion(in);
        }

        public Estacion[] newArray(int size)
        {
            return new Estacion[size];
        }
    };
```

Como se puede observar en el fragmento de código anterior, el campo *CREATOR* utiliza un constructor de la clase *Estacion* que recibe un objeto *Parcel* como parámetro. Este constructor también debe ser implementado y los campos de la nueva instancia deben reconstruirse a partir de los datos del *Parcel* recibido. El método *readFromParcel()* no forma parte de la interfaz pero se ha creado para mantener el código más claro y ordenado. El objeto *Parcel* no contiene claves ni índices para los datos introducidos en él; para recuperar los datos, éstos se deben leer en el mismo orden en el que fueron escritos.

```
      public Estacion(Parcel in)
      {
            this.readFromParcel(in);
      }


      private void readFromParcel(Parcel in)
      {
            this._id = in.readInt();
            this.nombre = in.readString();
            this.altitud = in.readInt();
```

```
            this.fechaInstalacion = in.readString();
            this.coordenadas =
                  in.readParcelable(LatLng.class.getClassLoader());
      }
```

Con esto finalizan las tareas de implementación de la interfaz *Parcelable*. En este caso, los objetos de la clase *Estacion* ya están capacitados para ser transferidos entre los diversos componentes de la aplicación. En el siguiente ejemplo se muestra la llamada para lanzar la actividad *DetalleEstacionActivity* desde otra actividad. En ella se utiliza el método *putExtra()* del *intent* para transferir el objeto estación. Este método está sobrecargado y admite parámetros de todos los tipos básicos, además de objetos que implementen las interfaces *Serializable* o *Parcelable*.

```
      private void verDetalleEstacion(Estacion e)
      {
            Intent i = new Intent(this, DetalleEstacionActivity.class);
            i.putExtra(Estacion.BUNDLE_EXTRA_NAME, e);

            this.startActivity(i);
      }
```

Y la siguiente línea de código permite recuperar la estación almacenada en el *intent*, desde la actividad de destino:

```
(Estacion) this.getIntent().getParcelableExtra(Estacion.BUNDLE_EXTRA_NAME);
```

### 5.2.2. Tareas asíncronas para la obtención de datos desde el servidor

Toda comunicación entre la aplicación cliente y el servidor se realiza exclusivamente mediante llamadas asíncronas, que son ejecutadas en un hilo diferente al de la interfaz de usuario de la aplicación. De esta forma se consigue evitar el bloqueo de la interfaz ante tareas de ejecución pesadas.

La implementación de las tareas asíncronas se ha realizado mediante la clase *AsyncTask* de la API de Android. Esta clase, pensada para ser extendida, permite la ejecución de tareas en segundo plano y la publicación de los resultados en el hilo de la interfaz de usuario, sin que el desarrollador tenga que preocuparse por la gestión de procesos e hilos de ejecución.

*AsyncTask* es una clase genérica, con tres parámetros, que definen: el tipo de parámetro recibido en la función *doInBackground()* encargada de realizar la ejecución en segundo plano, el tipo de datos de los valores que definen el progreso de la ejecución y el tipo de datos devuelto tras la ejecución en segundo plano.

```
android.os.AsyncTask<Params, Progress, Result>
```

Además cuenta con una serie de métodos de *callback*, lanzados antes y después de la ejecución en segundo plano, y tras cada actualización del progreso. A diferencia de *doInBackground()*, estos métodos se ejecutan en el hilo de la interfaz, permitiendo al desarrollador realizar modificaciones en ésta, para por ejemplo, mostrar una barra de progreso o notificar de alguna forma al usuario.

Al igual que ocurre con las actividades, se ha creado una clase base con todo el código común a cualquier llamada asíncrona realizada por la aplicación. En esta clase se realizan las tareas de codificación y decodificación de los datos enviados y recibidos, se configuran las cabeceras de HTTP pertinentes y se realiza un tratamiento genérico de los posibles errores producidos durante la ejecución de la tarea. Las llamadas asíncronas concretas realizadas desde cada actividad, se implementan mediante la creación de clases internas que extienden la clase base de tareas asíncronas, con el código específico de tratamiento de la respuesta obtenida desde el servidor:

```
private class CultivosTask extends BaseActivity.ActivityBaseTask
{

	@Override
	protected void onSuccesPostExecute(String result)
	{
		dictCultivos =
		 JSONUtils.jSONArrayToSparseArray(result, CultivoParcela.class);
		cargarDatosCultivos(dictCultivos);
	}

}
```

El ejemplo anterior corresponde a la clase interna que gestiona la llamada asíncrona de obtención de cultivos. Tras recibir los resultados del servidor, transforma los datos recibidos (en formato JSON) en objetos del modelo del sistema y posteriormente los carga en la interfaz de la actividad. A continuación se muestra el código necesario para lanzar esta llamada asíncrona desde la actividad:

```
CultivosTask ctask = new CultivosTask();
ctask.setRequest(new HttpGet());
ctask.execute( URL_SERVICE );
```

#### *5.2.2.1. Interfaz IParseFromJSON*

Para facilitar el paso entre los datos en formato JSON y los objetos del modelo de datos, se ha creado la interfaz *IParseFromJSON*. Esta interfaz ayuda a mantener ordenado todo el código de transformación, a la vez que proporciona un mayor grado de automatización a todo el proceso de cambio de formato. La interfaz debe ser implementada por cualquier objeto de la aplicación que tenga que ser transmitido desde el servidor a la aplicación.

```
public interface IParseFromJSON
{
	void getFromJSON(String strJSON);

	int get_id();
}
```

Como se puede apreciar, *IParseFromJSON* contiene dos métodos, por una parte *getFromJSON()* es el método donde el desarrollador debe realizar la transformación concreta del objeto a partir del *String* en formato JSON recibido. Con la función *get_id()* se obliga a que cada objeto transformado disponga de un identificador. Esto es útil cuando se leen listas de elementos obtenidas desde el servidor y se insertan automáticamente en un diccionario de datos, utilizando el identificador obtenido por *get_id()* como clave. En el siguiente ejemplo se muestra parte de la implementación de esta interfaz en la clase *Parcela*:

```
@Override
public void getFromJSON(String strJSON)
{
	try
	{
		JSONObject jObject = new JSONObject(strJSON);

		this.coordenadas = new ArrayList<LatLng>();

		this._id = jObject.getInt(Parcela.JSON_FIELD_ID);
		this.nombre =
```

```
                        jObject.getString(Parcela.JSON_FIELD_NOMBRE);

                …
        }
        catch (JSONException e)
        {
                Log.e("FarmCrop", e.getMessage());
        }
}


@Override
public int get_id()
{
        return _id;
}
```

### 5.2.3. Implementación de la interfaz

En esta sección se describe la implementación de los principales componentes que forman parte de la interfaz de FarmCrop. En primer lugar se habla de la creación de *layouts* como elemento básico de la interfaz, para seguidamente comentar vistas más específicas como los *ListViews* y sus posibles personalizaciones.

#### *5.2.3.1. Layouts*

Android proporciona dos formas de definir la disposición de los elementos de la interfaz de usuario: mediante recursos en forma de archivos XML y a través de la instanciación de los elementos de la interfaz directamente en el código. En esta sección se hablará exclusivamente de la definición de *layouts* a través de archivos XML, pues ha sido la opción más utilizada a lo largo del proyecto.

Para la definición de los *Views* y *ViewGroups* que forman parte de una interfaz, Android proporciona una serie de etiquetas XML que pueden ser anidadas para formar estructuras, de forma similar a como ocurre en el desarrollo de páginas web. Cada etiqueta representa a un *View* o *ViewGroup* en particular y, dependiendo del tipo de elemento, cuenta con una serie de parámetros que controlan diversos aspectos como el comportamiento, la apariencia o la distribución de los elementos de la vista.


**Ilustración 31: Interfaz de login**

En el siguiente ejemplo, perteneciente a una vista de tipo Button, se pueden apreciar algunos de los atributos comentados anteriormente. Cabe destacar el parámetro ID, representado por un número entero, que debe ser definido para cada vista del *layout*. Esto permite identificar cada elemento dentro de la jerarquía de vistas y además, proporciona una forma de referenciar los elementos desde cualquier otro lugar de la aplicación, incluyendo los recursos.

```
<Button
       android:id="@+id/buttonDescartar"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:background="@android:drawable/btn_default"
       android:text="@string/button_label_descartar" />
```

Cada subclase de *ViewGroup* proporciona una manera única de visualización y distribución de los elementos de la interfaz a los que engloba. A continuación se citan las subclases más habituales de *ViewGroup* pertenecientes a la API de Android. Todas han sido utilizadas, en mayor o menor medida, durante el desarrollo del proyecto:

- *LinearLayout*: Este *ViewGroup* organiza la distribución de sus elementos hijos en una sola fila, horizontal o vertical.
- *RelativeLayout*: Es el *layout* más flexible de todos. Permite ubicar los objetos hijos mediante referencias a otros elementos del *layout* o a los márgenes del propio contenedor.
- *WebView*: Sirve de contenedor para mostrar páginas web.
- *ListView*: Muestra una lista de elementos, generalmente dinámicos, en una sola columna.
- *ExpandableListView*: Similar a *ListView* pero con listas de dos niveles de profundidad.
- *TableLayout*: Distribuye sus elementos hijos en filas y columnas.

En el siguiente ejemplo se muestra simplificado el *layout* "*login_layout.xml*", perteneciente a la interfaz de la actividad *LoginActivity*. En él se puede apreciar la utilización de un *RelativeLayout* y un *TableLayout* entre otras vistas:

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/ScrollViewLogin"
    … >

    <RelativeLayout android:id="@+id/RelativeLayoutLogin" … >
        <ImageView android:id="@+id/imageViewLogo" … />
        <TableLayout android:id="@+id/tableLayoutLogin" … >

            <TableRow android:id="@+id/tableRow1" … >

                <EditText android:id="@+id/editTextUsuario" … >
                </EditText>

            </TableRow>

            <TableRow android:id="@+id/tableRow2" … >

                <EditText android:id="@+id/editTextClave" … >
                </EditText>

            </TableRow>

            <TableRow android:id="@+id/tableRow3" … >

                <Button android:id="@+id/buttonLogin" … />

            </TableRow>

        </TableLayout>
    </RelativeLayout>
```

```
</ScrollView>
```

Como ya se ha comentado en apartados anteriores, una vez definido el *layout* en un fichero XML de los recursos de la aplicación, éste puede cargarse llamando a *setContentView()* desde el método *onCreate()* de la actividad. La interfaz resultante tras la carga del *layout* del ejemplo anterior puede observarse en la Ilustración 31.

#### *5.2.3.2. Personalización de ListViews*

*ListView* es una vista que muestra una lista de elementos de forma dinámica, insertados a través de un *Adapter* que extrae el contenido de algún origen de datos, como un recurso de la aplicación, un vector de objetos o una consulta a una base de datos. Se trata de un elemento altamente personalizable y que puede dar lugar a vistas con una apariencia totalmente distinta.

Para una implementación básica de un *ListView*, Android proporciona diversos recursos y algunos *Adapters* genéricos para cargar datos desde distintos orígenes. Los más comunes son: *ArrayAdapter*, *CursorAdaptar* o *SimpleAdapter*. En el siguiente ejemplo se muestra la carga de una lista de estaciones en un *ListView* a través de un *ArrayAdapter*. En la creación del *Adapter* se indica el *layout* que se usará para cada celda, el *id* del elemento dentro del *layout* que contendrá el texto de cada estación y la lista de objetos *Estacion*. La vista resultante puede observarse en la figura de la izquierda de la Ilustración 32.

```
ArrayAdapter<Estacion> adapter = new ArrayAdapter<Estacion>(
       this,
        android.R.layout.simple_list_item_1,
        android.R.id.text1,
        arrayEstaciones);

this.listView.setAdapter(adapter);
```

Existen diversas opciones a la hora de realizar implementaciones más personalizadas de un *ListView*. En este proyecto se ha desarrollado una aproximación que permite la creación de *ListViews* con filas heterogéneas, que cuentan con interfaces y comportamientos distintos.

Para realizar esta implementación, se ha creado una subclase de *BaseAdapter* (clase abstracta de la API de Android con funcionalidades comunes de los *adapters*) y

se han sobrescrito los métodos básicos de la interfaz *Adapter*. En el siguiente ejemplo se muestra parte de esta implementación:

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


	@Override
	public Object getItem(int position)
	{
		return this.items.get(position);
	}


	@Override
	public long getItemId(int position)
	{
		return this.items.get(position).getId();
	}
```

Para lograr que cada ítem del *ListView* cuente con su interfaz y su comportamiento propios, se ha creado una estructura de ítems, asociada al *adapter*, con un tipo base: *BaseItem*. Esta clase raíz cuenta con la función abstracta *makeView()*, que toda subclase debe implementar. En esta función es donde cada ítem concreto describe cómo debe ser su interfaz. El siguiente fragmento de código corresponde a la implementación de *makeView()* en la clase *EditTextItem*:

```
@Override
public View makeView(LayoutInflater inflator, View v)
{
	v = (v == null) ? inflator.inflate(this.getLayout(), null) : v;

	((TextView) v.findViewById(this.textViewId)).setText(this.getText());
	((TextView) v.findViewById(R.id.textViewUnit)).setText(this.unit);

	this.editText = (ExtendedEditText) v.findViewById(R.id.editTextValue);
	this.editText.clearTextChangedListeners();
	this.editText.addTextChangedListener(this.textWatcher);
	this.editText.setText(this.value);

	return v;
}
```

De esta forma, cuando el *adapter* necesita crear la interfaz de una fila, en lugar de crearla él mismo, accede a su lista interna de *BaseItems*, busca el ítem asociado a la fila en cuestión y ejecuta la función *makeView()* del ítem:

```
@Override
public View getView(int position, View convertView, ViewGroup parent)
{
	return this.items.get(position).makeView(inflator, convertView);
}
```

En la Ilustración 32 se muestran algunos de los *ListViews* desarrollados en la aplicación. El primero de ellos corresponde a una implementación simple mediante un *ArrayAdapter*. En el *ListView* del centro puede observarse la estructura de ítems heterogéneos descrita anteriormente. La vista contiene tres ítems diferentes: uno con solo texto, otros con un enlace navegable y un tercero que muestra una imagen desde una dirección en internet (en este caso de Google Static Maps). El tercer *ListView* contiene una lista homogénea de elementos pero con una interfaz personalizada.

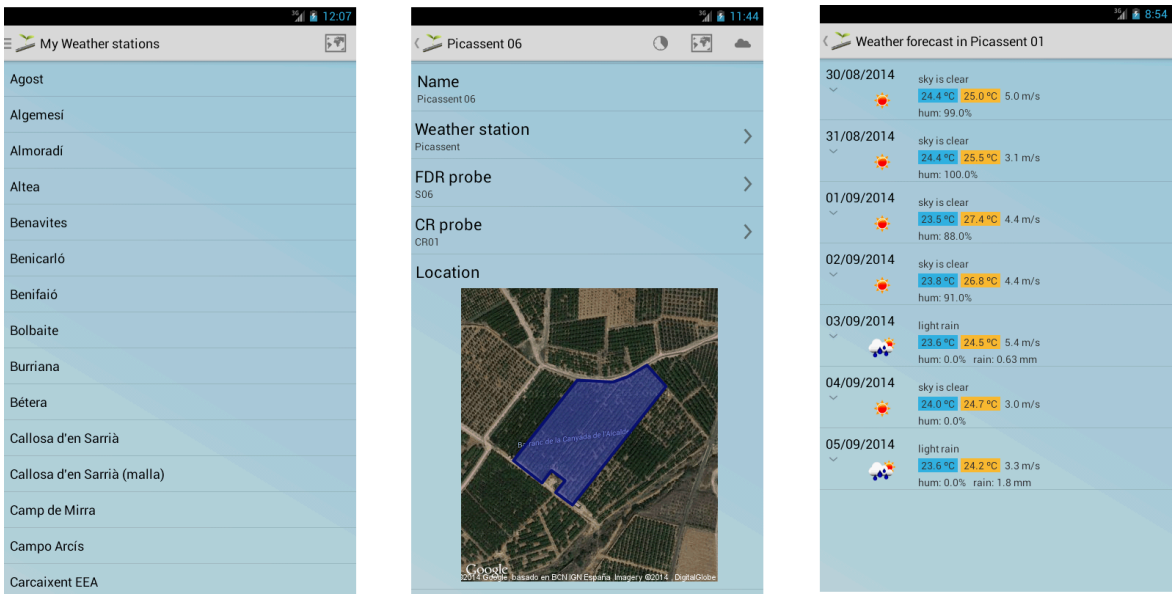



**Ilustración 32: *ListViews* en FarmCrop**

# 6. Conclusiones

En el presente trabajo final de máster se ha desarrollado un sistema, con una arquitectura orientada a servicios, que encapsula el acceso a una herramienta de cálculo de necesidades de riego para parcelas agrícolas. También se ha desarrollado una aplicación para la plataforma móvil Android que hace uso de los servicios proporcionados por el sistema.

La aplicación desarrollada cuenta con un mayor grado de usabilidad, es más sencilla de utilizar y cuenta con mayores prestaciones que la herramienta de cálculo de riego original. Además, el desarrollo no se ha centrado únicamente en el módulo de cálculo del riego, sino que se han implementado otras funcionalidades, coherentes con el contexto del sistema, y que en conjunto conforman una completa aplicación donde el usuario puede definir sus parcelas, acceder a la predicción meteorológica o ver las mediciones de estaciones y sondas.

La arquitectura del sistema ha sido desarrollada para que resulte extensible, escalable y robusta. La capa de servicios desarrollada en el marco de este trabajo ya ha sido utilizada como modelo para nuevos proyectos desarrollados en el IIAMA, cumpliendo así uno de los objetivos iniciales.

Sin embargo, por el momento no se ha podido completar el objetivo de dar mayor repercusión y visibilidad a la aplicación de cálculo de riego. A fecha de 16/08/2014 la primera versión de la aplicación (con las funcionalidades expuestas en este documento a excepción del módulo de sondas) está completada y lista para ser publicada, pero el instituto está a la espera de que desde el Ministerio de Agricultura, Alimentación y Medio Ambiente autoricen la utilización de los datos de sus estaciones de la red SIAR.

En el caso de que el Ministerio no autorice al IIAMA para que pueda utilizar su red de estaciones meteorológicas, implantadas con el objetivo de luchar contra la sequía, en una aplicación que indirectamente favorece el ahorro de agua (ya se han dado casos similares), el sistema puede cambiar su origen de datos climáticos sin excesivos esfuerzos de desarrollo. Esto es debido a la filosofía de desarrollo modular y de componentes poco acoplados entre sí que se ha seguido a lo largo de todo el proceso de desarrollo. Si bien es cierto que las estaciones de la red SIAR ofrecen un grado ideal de idoneidad para el proyecto, por su ubicación en lugares estratégicos de explotaciones agrícolas, no son el único servicio disponible para la obtención de los parámetros climáticos necesarios para realizar el cálculo de las necesidades hídricas.

En cuanto a las tecnologías utilizadas, el desarrollo de la aplicación para Android ha sido, en ocasiones, bastante complejo. No tanto desde el punto de vista técnico, sino que más bien ha sido como montar un gran puzle. Por ejemplo, hacer que la aplicación fuese compatible con las versiones anteriores de Android ha sido complicado en algunas fases del desarrollo, con multitud de métodos sin poder ser utilizados por pertenecer a una versión mayor a la mínima soportada, otros métodos catalogados como obsoletos, o comportamientos "extraños" de la aplicación según modelo del dispositivo o versión del SO.

Otro de los inconvenientes del desarrollo de aplicaciones para Android (si bien también es uno de sus grandes atractivos), es la ingente cantidad de dispositivos y tamaños de pantalla diferentes existentes en el mercado, algo que no ocurre por ejemplo en el desarrollo para iOS. Pese a todo, la experiencia de trabajo con el SDK de Android ha sido positiva y muy didáctica.

Por otro lado, el desarrollo de la parte del servidor ha sido relativamente más sencillo. La forma de trabajar con Web API en la definición de servicios y rutas es intuitiva y potente, y en lo que respecta a la base de datos, las funciones geoespaciales incluidas en MS SQL Server han sido también de gran ayuda. Además, las características que llevan integradas los lenguajes de la plataforma .NET, como LINQ, funciones anónimas, reflexión, etc. facilitan en gran medida la expresividad del programador y bien utilizadas, generan código más limpio, más compacto y con un mayor grado de optimización.

Por último, a nivel personal, el presente trabajo final de máster ha sido de gran utilidad para adquirir nuevos conocimientos, tanto en las tecnologías comentadas anteriormente, como en muchos de los procesos técnicos que tienen lugar en una explotación agrícola.

## 6.1. Trabajo futuro

El sistema desarrollado en este trabajo no es un proyecto cerrado. Existen diversos desarrollos en activo así como nuevos caminos interesantes que pueden ser candidatos potenciales para futuras implementaciones. A continuación se describen algunas de las nuevas tareas de desarrollo a tener en cuenta:

**Nuevas funcionalidades relacionadas con la geolocalización de parcelas:** En el punto de desarrollo actual, la localización de las parcelas a través de la aplicación móvil es una característica más visual que funcional. La geolocalización de las parcelas puede ser explotada de forma más exhaustiva en la aplicación, para aportar nuevas funcionalidades interesantes. Por ejemplo, mediante la combinación de la localización del perímetro de la parcela con imágenes vía satélite y empleando los algoritmos de

tratamiento de imágenes adecuado, se puede calcular de forma precisa el porcentaje de área sombreada, sin que sea necesario que este dato lo proporcione el agricultor manualmente.

**Comparación entre agua proporcionada y recomendación de riego:** En el futuro, podría implementarse la conexión con algunos de los sistemas en tiempo real ya existentes en algunas comunidades de regantes. Con esta conexión, entre otros datos, se podría obtener la planificación de riego y el volumen proporcionado a cada parcela de la comunidad. El sistema podría proporcionar al usuario todos estos datos y realizar interesantes comparativas, por ejemplo, entre el agua proporcionada a una parcela y la recomendación de riego para la parcela en cuestión.

**Integración completa con sondas de humedad:** Si se desarrolla el módulo de conexión con sondas de humedad, un usuario que contase con estos dispositivos en sus parcelas podría conectarlas al sistema y realizar los cálculos de riego por otros métodos (ver apartado 2.1). Una vez hecho esto, podría comparar ambos resultados y aplicar el que considere más conveniente.

**Desarrollo de la aplicación cliente para otras plataformas:** Puede resultar interesante la adaptación de la aplicación de Android a otras plataformas móviles como iOS o Windows Phone. De forma complementaria también puede plantearse la creación de una aplicación web que proporcione mejores opciones de visualización e interacción frente a los recursos limitados de los dispositivos móviles.

# 7. Referencias

1. Instituto de Ingeniería del Agua y Medio Ambiente. http://www.iiama.upv.es.

2. Instituto Valenciano de Investigaciones Agrarias (IVIA). http://www.riegos.ivia.es.

3. AGADAPT. http://www6.inra.fr/agadapt.

4. Máster en Ingeniería de Software, Métodos Formales y Sistemas de Información. http://www.upv.es/titulaciones/MUISMFSI/indexc.html.

5. Departamento de Sistemas Informáticos y Computación. https://www.upv.es/entidades/DSIC/index.html.

6. Universitat Politècnica de València. http://www.upv.es.

7. Herramienta de cálculo de riego del IVIA. http://riegos.ivia.es/calculo-de-necesidades-de-riego.

8. Android. http://www.android.com/.

9. **Bello, Miguel Ángel Jiménez.** Integración de los procesos agronómicos e hidráulicos del riego a presión en un entorno SIG para la gestión eficiente de Comunidades de Regantes. 2008, Capítulo 4.

10. *Nuevas técnicas para el manejo de riego de los cítricos.* **Intrigliolo D., Bonet L., Castel J.R.** 37, 2008, Agrónomos, págs. 48-57.

11. *Crop evapotranspiration. Guidelines for computing crop water requirements.* **Allen R. G., Pereira L. S. y Raes D.** 56, 1998, FAO Irrigation and drainage. FAO - Food and Agriculture Organization of the United Nations, Rome.

12. *Riego Localizado. Diseño de instalaciones.* **Montalvo, T.** 2002. Universidad Politécnica de Valencia.

13. Interreg II C. http://ec.europa.eu/regional_policy/archive/interreg3/inte2/inte2c.htm.

14. SIAR. http://eportal.magrama.gob.es/websiar/Inicio.aspx.

15. Open Handset Alliance. http://www.openhandsetalliance.com/.

16. SQLite. http://www.sqlite.org/.

17. Dalvik. http://source.android.com/devices/tech/dalvik/.

18. Google Play. https://play.google.com/store.

19. .NET. http://www.microsoft.com/net.

20. Visual Studio. http://www.visualstudio.com/.

21. ISO. http://www.iso.org.

22. Ecma International. http://www.ecma-international.org/.

23. Mono. http://www.mono-project.com.

24. ASP.NET. http://www.asp.net/.

25. **Fielding, Roy Thomas.** Architectural Styles and the Design of Network-based Software Architectures. 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

26. OpenWeatherMap. http://openweathermap.org/.

27. Google Maps. https://developers.google.com/maps/?hl=es.

28. Navigation Drawer. https://developer.android.com/design/patterns/navigation-drawer.html.

29. Balsamiq. https://balsamiq.com/.

30. Android Developers Tools. http://developer.android.com/tools/index.html.

31. ASP.NET Web API. http://www.asp.net/web-api.

32. ASP.NET MVC. http://www.asp.net/mvc.

33. *HTTP Authentication: Basic and Digest Access Authentication.* **Franks, et al.** 1999.

34. OpenID. http://openid.net/.

35. OAuth. http://oauth.net/.

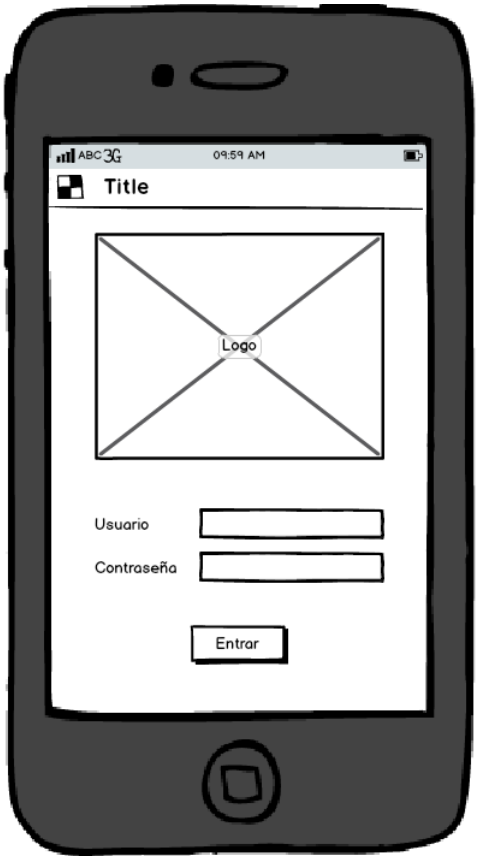
36. NHibernate. http://nhforge.org/.

37. Entity Framework. http://msdn.microsoft.com/es-es/data/ef.

38. **Smith, Jeff.** Implementing Table Inheritance in SQL Server. 2008. http://www.sqlteam.com/article/implementing-table-inheritance-in-sql-server.

39. Android SDK. http://developer.android.com/sdk/index.html.

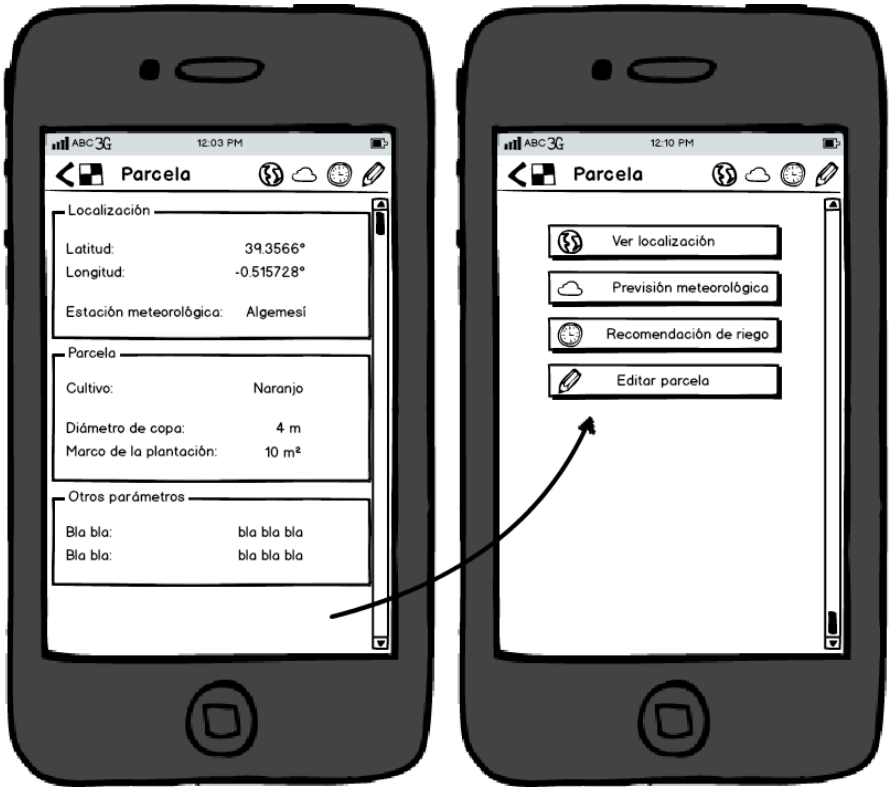
# I. Anexo: Bocetos de interfaces de usuario




**Interfaz de login**


**Menú principal**


**Lista de parcelas**



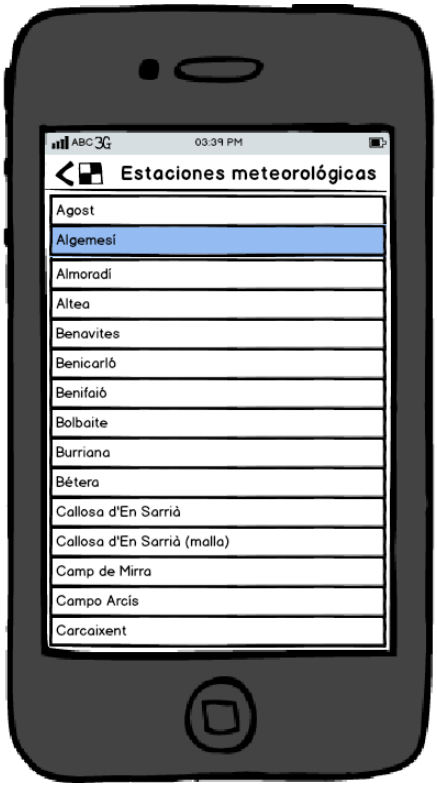

**Detalle de una parcela**


**Localización geográfica de estaciones y parcelas**


**Previsión meteorológica**


**Formulario de edición y creación de parcelas**


**Recomendación de riego**




**Lista de estaciones**


**Detalle de una estación**


**Mediciones de una estación**

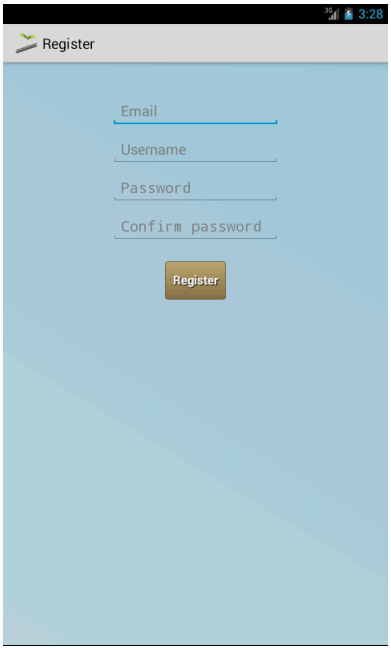
# II. Anexo: Manual de usuario

La aplicación FarmCrop permite al usuario la definición de sus parcelas mediante parámetros técnicos así como la ubicación geográfica de las mismas. Posteriormente el usuario puede realizar acciones como: calcular las necesidades de riego de sus cultivos, consultar las mediciones de las estaciones meteorológicas cercanas a sus parcelas, consultar la previsión meteorológica para los próximos días, etc. En este anexo se detallan las instrucciones necesarias para llevar a cabo correctamente todas estas acciones.

## Registro y autentificación

Al iniciar la aplicación FarmCrop, se accede en primer lugar, a la pantalla de *login*. Desde esta pantalla el usuario puede introducir sus credenciales o acceder a la pantalla de registro en el caso de que aún no posea una cuenta en el sistema. La aplicación recordará los últimos datos de autentificación introducidos, para evitar que el usuario tenga que volver a teclearlos.


**Pantalla de *login***

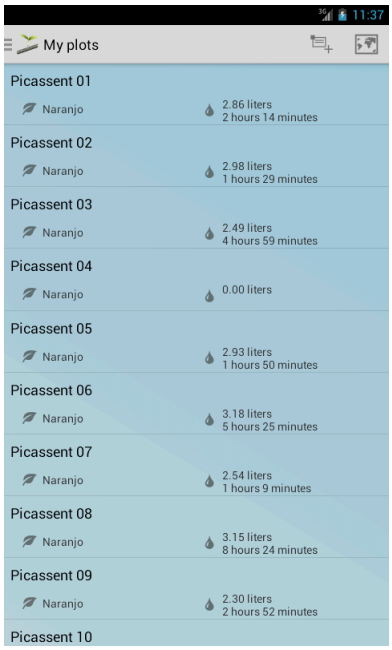



**Pantalla de registro**

Desde la pantalla de registro el usuario puede fácilmente crear una nueva cuenta mediante la introducción de unos pocos campos. Los datos de la nueva cuenta se almacenan en los servidores del IIAMA e inmediatamente el usuario ya está capacitado para acceder a la aplicación.

### Lista de parcelas

Tras identificarse, el usuario accede a la pantalla inicial de la aplicación, que muestra la lista de todas las parcelas definidas. Desde esta pantalla el usuario puede previsualizar el tipo de cultivo de cada parcela, así como la recomendación de riego para el periodo de tiempo que se haya establecido desde la configuración de la aplicación. Además, desde esta pantalla el usuario puede acceder a otras acciones como definir una nueva parcela o ver el mapa con la ubicación de todas las parcelas.

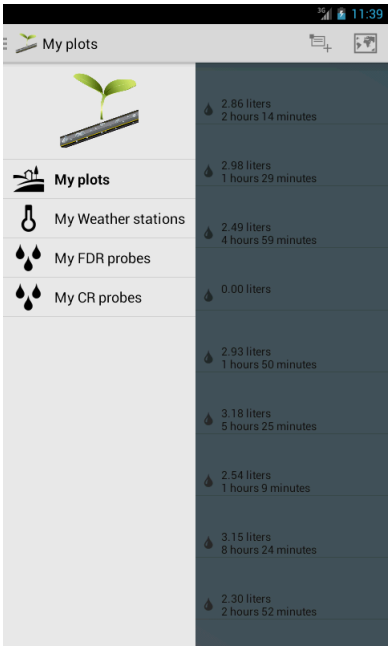



**Lista de parcelas**


**Mapa con la ubicación de las parcelas**

### Menú lateral

Desde la pantalla inicial, el usuario puede desplegar el menú lateral y navegar a cualquiera de las otras vistas principales de la aplicación. Para abrir este menú, basta con pulsar en el icono de la parte superior izquierda de la pantalla, o bien realizar un gesto de arrastre desde el extremo izquierdo de la pantalla hacia el centro. Todas las pantallas principales de la aplicación cuentan con este menú de navegación.

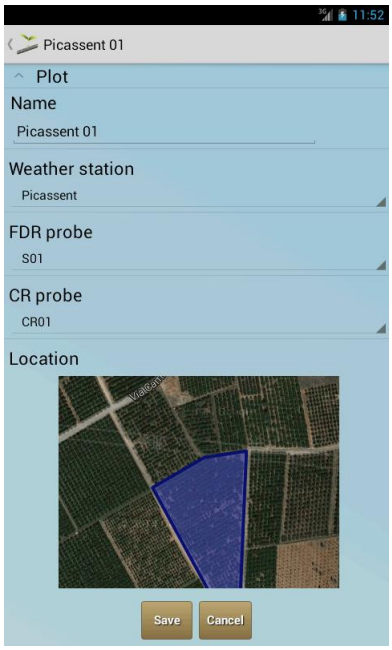



**Menú lateral**

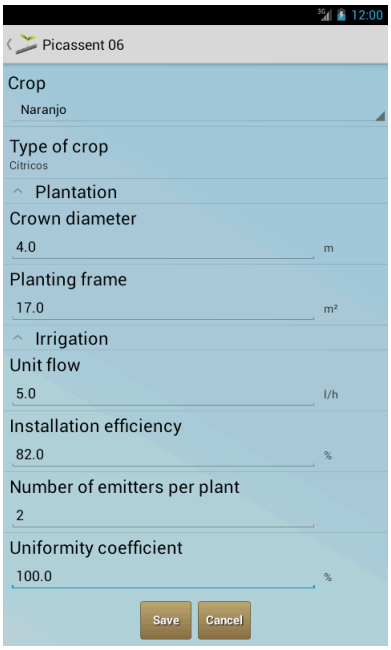
### Definición de parcelas

Para definir una nueva parcela el usuario debe dirigirse a la pantalla con la lista de todas las parcelas, para posteriormente pulsar en el botón “Nueva parcela” de la *Action Bar*. De esta forma accederá al formulario de creación de nuevas parcelas donde deberá introducir la mayor cantidad posible de parámetros técnicos (cuantos más parámetros se incluyan, más precisa será la recomendación de riego proporcionada por la aplicación). Desde este formulario también se puede asociar a la parcela la estación meteorológica más cercana, o bien otra distinta a criterio del usuario.

Los parámetros técnicos se dividen en tres grupos y son diferentes según se trate de un cultivo hortícola o un cultivo frutal, aunque actualmente FarmCrop sólo contempla la especificación de cultivos frutales. En primer lugar, en el apartado de “Plantación” se definen tanto el diámetro de medio de copa como el marco de plantación, en metros cuadrados, que suelen ocupar los frutales. En el apartado de “Instalación de riego” se deben introducir entre otros aspectos, el número de emisores de goteo por planta y el caudal unitario de los emisores. Por último, en “Parámetros auxiliares” se encuentran los campos de salinidad del suelo y coeficiente de parcela (coeficiente reductor de las dotaciones de riego por acolchados de plástico u otros elementos).




**Formulario de edición I**



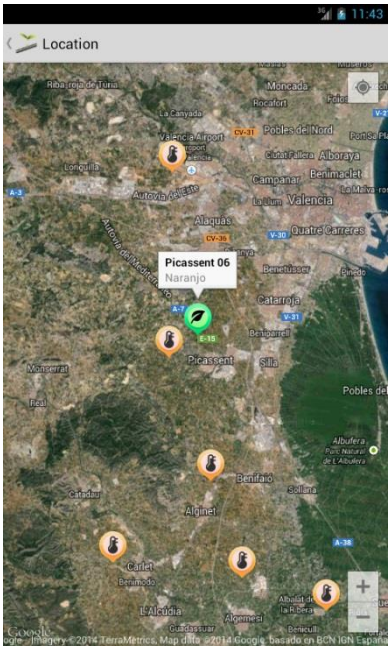

**Formulario de edición II**

Desde el formulario de edición, el usuario tiene acceso a una vista desde donde puede definir la ubicación de la parcela a través de un mapa. La localización de la parcela se puede realizar, bien mediante la definición de un punto aproximado sobre la ubicación de la parcela, o bien definiendo su perímetro completo a través de un polígono dibujado sobre el mapa.

Para realizar alguna de estas dos acciones, el usuario debe efectuar una pulsación larga sobre el mapa en la localización de su parcela. Entonces la aplicación mostrará un menú contextual preguntando al usuario sobre “Ubicar la parcela en este punto” o “Iniciar la definición del perímetro de la parcela”. Si se elige la primera opción, la aplicación creará una marca en el mapa, en el lugar especificado y con los datos básicos de la parcela. Si se elige la segunda opción, la vista entrará en “modo edición” y permitirá al usuario definir un polígono sobre la superficie del mapa. Para delimitar el perímetro de la parcela, el usuario debe pulsar sobre cada uno de los vértices de la misma de forma correlativa (no importa si es en un sentido o en otro, siempre que los vértices se pulsen de forma ordenada) y posteriormente pulsar en “Hecho” si está satisfecho con el polígono definido.

Desde esta vista el usuario puede observar las estaciones meteorológicas más cercanas y asociar a su parcela la que crea más conveniente, solo con pulsar sobre la marca de la estación y posteriormente en “Asignar estación”. Las mediciones de la estación asociada se utilizarán para realizar el cálculo de las necesidades hídricas. Una vez que el usuario haya terminado de definir la localización de la parcela y la asignación de la estación, debe pulsar en el botón de “Confirmar ubicación” de la *Action Bar* para guardar los cambios realizados.

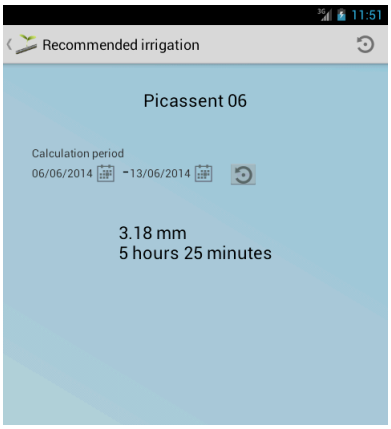

**Definición del perímetro de la parcela**




**Localización de la parcela**

## Recomendación de riego de una parcela

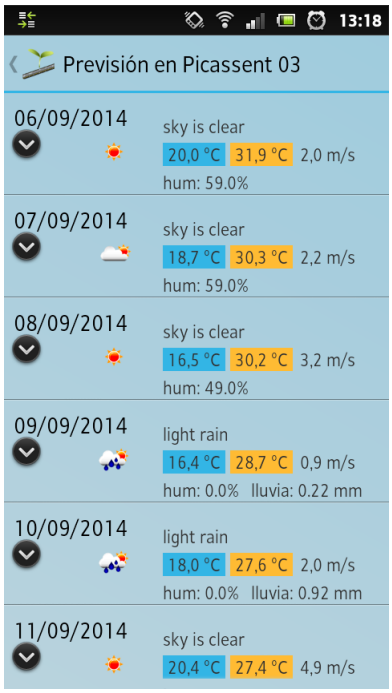
La recomendación de riego para una parcela se puede consultar, bien mediante la previsualización por defecto de la pantalla inicial, o bien a través de la vista específica de recomendación de riego. El usuario puede acceder a esta vista desde los detalles de la parcela; o bien desde la lista de parcelas, abriendo el menú contextual de una de ellas mediante una pulsación larga y eligiendo la opción correspondiente.



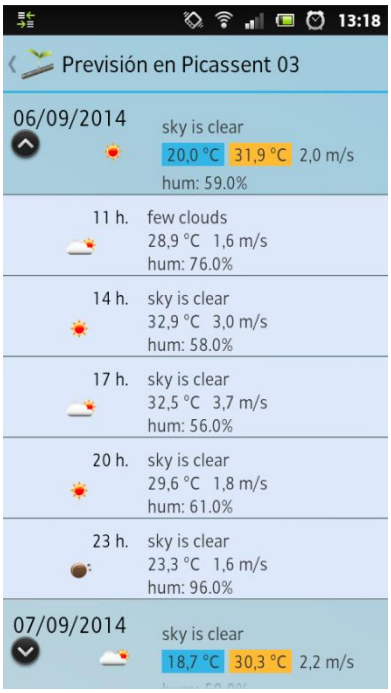

**Recomendación de riego**

### Previsión meteorológica

La aplicación dispone de una vista que muestra la previsión meteorológica para los próximos siete días, personalizada para la ubicación de la estación asociada a la parcela. La previsión tendrá una precisión de tres horas, salvo en los últimos días de la predicción, donde la precisión será diaria. A esta vista se puede acceder a través del menú de la pantalla donde se muestran los detalles de la parcela.




**Previsión meteorológica I**




**Previsión meteorológica II**

### Mediciones de las estaciones meteorológicas

FarmCrop cuenta con un módulo que representa, en forma de gráficos, las mediciones más representativas producidas en las estaciones meteorológicas incluidas en el sistema. El usuario puede llegar a esta vista a través del menú lateral, pulsando en “Estaciones meteorológicas” y posteriormente en la estación que quiera consultar. Una vez en la vista de mediciones, el usuario puede desplazarse entre los diversos gráficos a través de gestos de desplazamiento horizontal.

Los datos mostrados corresponden a las últimas mediciones realizadas, con el intervalo de tiempo especificado en la configuración de la aplicación (un día, una semana o un mes). Adicionalmente también es posible acceder a los datos históricos, seleccionando la opción correspondiente en el menú de la vista de mediciones. Los

gráficos mostrados en esta pantalla son los siguientes: temperatura, humedad, dirección del viento, precipitación y evapotranspiración (ETo).

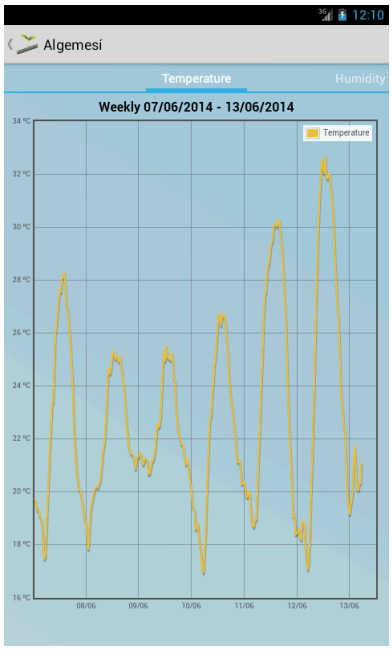



**Gráfico de temperatura. Intervalo semanal**

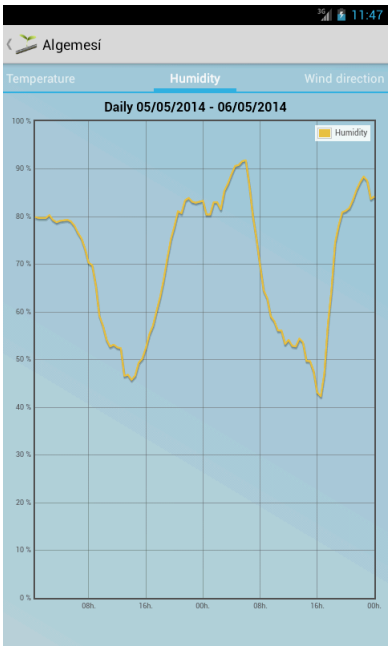
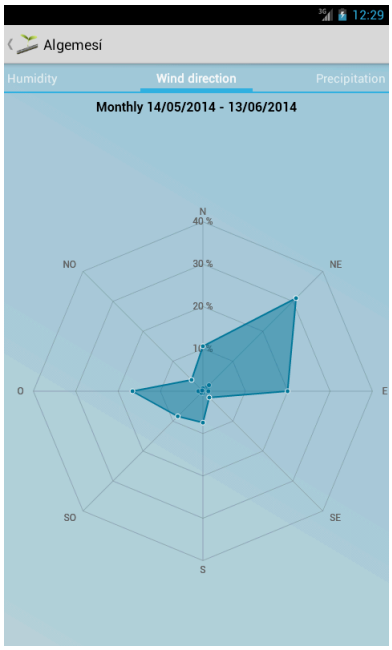
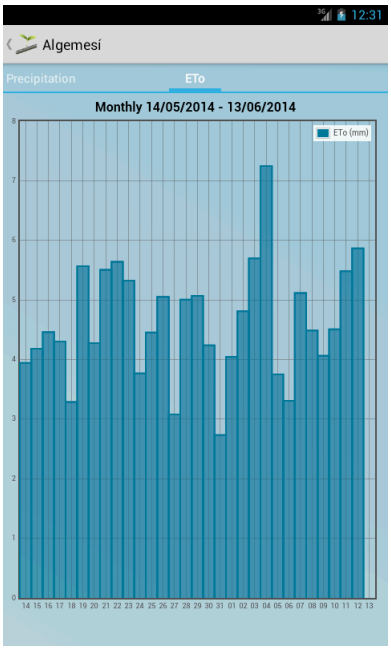



**Gráfico de humedad. Intervalo diario**




**Dirección del viento. Intervalo mensual**

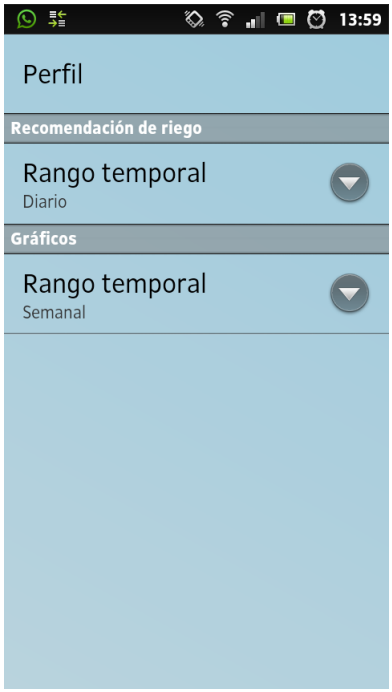



**ETo. Intervalo mensual**

### Configuración de la aplicación

El usuario puede acceder a la configuración de la aplicación a través del menú presente en cualquier vista. Desde la pantalla de configuración, puede establecerse el rango temporal de las recomendaciones de riego automáticas de la pantalla inicial y también, el rango temporal de las mediciones de las estaciones.

Por otro lado, la aplicación se encuentra traducida al español y al inglés, siendo éste último el idioma por defecto. Para seleccionar que idioma va a ser utilizado, la aplicación hace uso de la configuración regional del dispositivo.




**Pantalla de configuración**


**Rango temporal de las mediciones**