Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

# Introducción de aspectos de geolocalización en una vivienda inteligente

Trabajo Fin de Grado

**Grado en Ingeniería Informática**

**Autor**: Lorena Calabuig Monerris

**Tutor**: Joan Josep Fons i Cors

Septiembre 2014

*A mi familia, y en especial a mis padres,*

*que me han apoyado durante todos estos años.*

# Resumen

En este trabajo se van a introducir aspectos de geolocalización de los habitantes de una vivienda inteligente para mejorar el comportamiento de ésta y fomentar el ahorro energético. Para tal fin se va a desarrollar una aplicación móvil en Android que enviará la posición de cada habitante al servidor de la vivienda inteligente mediante peticiones REST y así pueda activar o desactivar los distintos dispositivos de la casa, como puede ser la climatización; además, avisará al usuario de la cercanía de supermercados en caso de tener que comprar alimentos.

**Palabras clave:** vivienda inteligente, geolocalización, rest, android.

# Abstract

In this paper geolocation aspects of the smart home's residents are be introduced to improve the behavior of house and encourage energy saving. To do this an Android mobile application have been developed that sends the position of each resident to smart home server via REST requests, so it can activate or deactivate the different devices in the house, as may be the climate; also the user will be notified of nearby supermarkets in case that user has to buy food.

**Keywords:** smart home, geolocation, rest, android.

# Tabla de contenidos

# Tabla de figuras

# 1. Introducción

En este capítulo se introducirá el problema que se desea resolver mediante una breve descripción, la motivación surgida a partir del contexto actual para la elección y la realización de este trabajo y los objetivos propuestos a cumplir durante el desarrollo del proyecto.

## 1.1. Descripción del problema

Las tecnologías de información y comunicación, así como la automatización, se encuentran en pleno auge y cada vez son más las personas que se suman a consumirlas y explotarlas. El hecho de aprovecharlas dentro del domicilio, nos conduce al concepto de domótica e incluso un poco más allá: hacer que nuestra vivienda sea autónoma, de forma que no resulte necesario preocuparse por hacer ciertas tareas rutinarias como apagar la climatización cuando se sale de casa.

La automatización de estos servicios lleva a la idea de vivienda inteligente o *smarthome*, permitiendo al usuario no tener que responsabilizarse del control de su casa delegándolo en un sistema y haciendo su vida diaria un poco más sencilla; además, fomenta el ahorro energético al no consumir electricidad o gas innecesario cuando no hay nadie en la vivienda.

## 1.2. Motivación

La realización de este proyecto ha sido posible por el interés que despierta el desarrollo de servicios que puedan controlar las diferentes funcionalidades de la casa de forma automática y del desarrollo de aplicaciones para dispositivos móviles que, además, servirán para controlar dichas funcionalidades de una forma segura y eficiente.

## 1.3. Estado del arte

El desarrollo de dispositivos “inteligentes” que permiten, de una forma más sencilla, interactuar con el usuario y facilitarle una tarea específica es una tecnología próspera; para ello, es imprescindible el uso de las TIC (Tecnologías de la Información y las Comunicaciones).

Con las *smarthomes*, la domótica da el salto hacia la automatización de los procesos que suceden dentro de la vivienda para facilitar la interacción con los habitantes de la casa. Si a esto se suma la integración de la posición de los usuarios para mejorar la automatización de estos procesos, se obtiene la optimización de la gestión energética de los recursos de un hogar y se proporciona al usuario la mínima interacción con dichos procesos, transmitiéndole tranquilidad y comodidad.

## 1.4. Objetivos

Este trabajo pretende desarrollar una aplicación móvil que envíe al servidor que controla los dispositivos de la casa, la posición de los miembros de la unidad familiar a intervalos de tiempo definidos, con el propósito de estudiar su comportamiento y así permitir encender y/o apagar los diferentes dispositivos de la casa para fomentar el ahorro energético cuando el usuario está fuera de casa, sin necesidad de que el usuario intervenga directamente.

Con tal fin, se estudiarán los requisitos funcionales tanto de la aplicación cliente como del servidor y se implementarán utilizando dos tecnologías en auge: Android y un servicio REST, respectivamente.

Las reglas a aplicar en cada momento se seleccionan automáticamente a través de un algoritmo de análisis de comportamiento del usuario que se implementará; se entiende regla como a un conjunto de acciones a realizar sobre los dispositivos de la casa.

## 1.5. Organización del trabajo

El presente trabajo se divide en 6 capítulos. En el capítulo 1 se expone una brevemente la descripción del contexto tecnológico actual, así como de la motivación y de los objetivos que se pretenden conseguir. El capítulo 2 resume las tecnologías y herramientas utilizadas para el desarrollo de este proyecto. El caso de estudio de este proyecto se expone en el capítulo 3 y que posteriormente se analiza en el capítulo 4, especificando los requisitos, modelos y estructuras de las aplicaciones cliente y servidor a implementar. El capítulo 5 hace referencia al diseño de la solución del problema planteado en el capítulo 4. Tras abordar los temas de análisis y diseño de la solución, en el capítulo 6 se explicará la implementación del servidor y la de la aplicación móvil cliente. Por último, en el capítulo 7 se recogen las conclusiones obtenidas y las posibles mejoras y ampliaciones del trabajo realizado.

# 2. Contexto tecnológico

En este apartado se describirán todas las tecnologías empleadas para el desarrollo e implementación tanto de la parte del servidor como la parte de la aplicación móvil que actuará como cliente.

## 2.1. Eclipse Luna

El entorno de desarrollo integrado (IDE) que se ha empleado para implementar y probar el código del servidor ha sido Eclipse Luna, que es la última versión estable hasta la fecha de esta plataforma y permite el desarrollo de aplicaciones con JavaSE/EE.

## 2.2. Java

El lenguaje de programación Java es un lenguaje imperativo multiplataforma de fácil portabilidad que tiene un extenso catálogo de librerías disponibles de uso libre y gratuito. Se ha utilizado este lenguaje porque se utilizarán dos tecnologías – Restlet y Android – que se basan en él.

## 2.3. Restlet

Restlet es un framework ligero de software libre desarrollado en Java que sigue la arquitectura REST y se beneficia de su simplicidad y escalabilidad. Se utiliza para crear servicios web RESTful, es compatible con todos los conceptos REST *(Resource*, *Representation*, *Component*, etc.) y soporta la mayoría de los estándares web como son HTTP, XML, JSON o SMTP, entre otros.

No es necesario tener unos conocimientos completos de los métodos, cabeceras y estados de HTTP, ya que Restlet proporciona negociación de contenido, compresión automática, representaciones parciales y procesamiento parcial (partiendo de ciertas condiciones iniciales).

Restlet además es multiplataforma y puede ser desplegado en JavaSE/EE, Google Web Toolkit, Google App Engine (GAE), OSGI y Android. Se ha utilizado la versión 2.2 de la API de Restlet, que es la última versión estable.

### 2.3.1. Arquitectura

El framework está dividido en dos partes principales: la primera está compuesta por el API de Restlet, que soporta los conceptos REST y HTTP, lo que facilita el manejo de llamadas tanto para aplicaciones cliente como aplicaciones servidor; la segunda capa está compuesta por el *Restlet Engine*, que es la implementación del API.

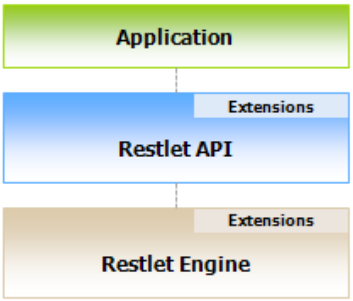



**Figura 1 - Arquitectura de Restlet**

### 2.3.2. Conceptos

A continuación se van a describir brevemente los conceptos REST más representativos que soporta Restlet:

- ***Application***: contiene la lógica de la aplicación y gestiona las peticiones HTTP mediante el enrutamiento (clase `Router`) de las mismas a los correspondientes recursos (*resources*).
- ***Component:*** es un contenedor de una aplicación Restlet y permite desplegarla como stand-alone. En el *Component* se indica el tipo de servicio que se va a desplegar (HTTP, SMTP, etc.) y el puerto; en caso de que se vayan a realizar peticiones a otro servidor, hay que definir también un cliente en este punto.
- ***Resource***: es donde se integran los objetos/recursos del dominio de la aplicación web. Sobre estos recursos se pueden atender las peticiones HTTP de GET, PUT, DELETE, POST y OPTIONS. Sólo el método GET está disponible por defecto. Hay dos tipos de recursos: para la parte cliente (`ClientResource`), que consume los recursos del servidor; y para la parte servidora (`ServerResource`), que pone a disponibilidad de los clientes los recursos disponibles en el servidor.
- ***Representation***: es el estado actual de un recurso y puede estar representado por cualquier estándar web, como puede ser JSON o XML.

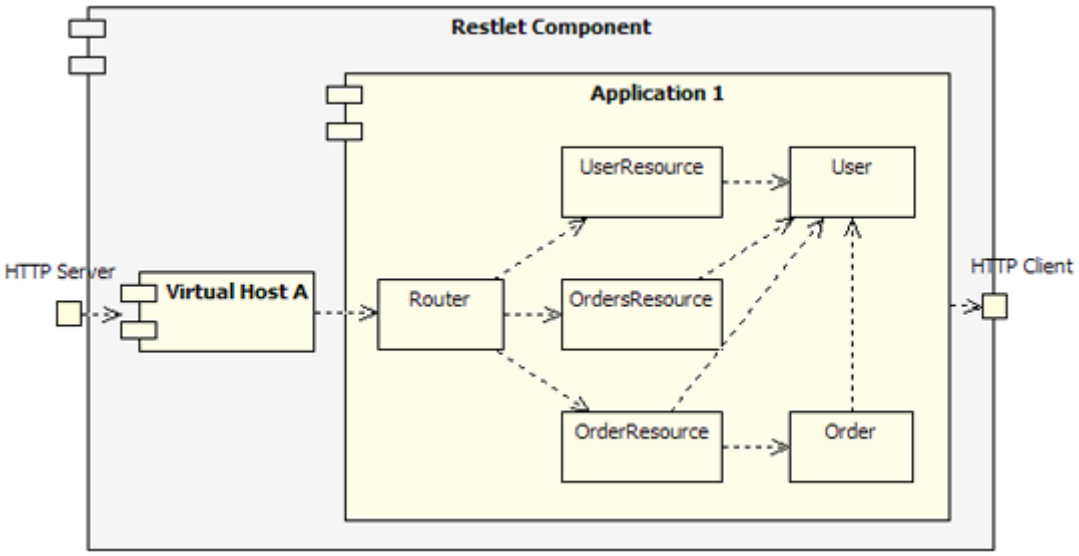



**Figura 2 - Ejemplo de relación entre los distintos de Restlet**

## 2.4. Android Studio

Para el desarrollo de la aplicación móvil se ha utilizado el IDE Android Studio, desarrollado por Google, que está especialmente diseñado para el desarrollo de aplicaciones Android; según Google será el IDE oficial para Android cuando terminen de desarrollar una versión estable – ahora mismo está en fase beta –.

Proporciona ciertas ventajas frente a Eclipse ADT (Android Development Tools) como una mejora de la interfaz para el desarrollo de layouts, reutilización de código, o una mejor gestión de las dependencias.

## 2.5. Android OS

Android es un sistema operativo para dispositivos móviles de código abierto basado en el kernel de Linux que está siendo desarrollado por Google.

Se ha decidido desarrollar en esta plataforma la aplicación porque es uno de los sistemas operativos para smartphones y tablets con mayor cuota de mercado hasta la fecha. A continuación se muestra un diagrama de tarta con la cuota de mercado en agosto de 2014 de los diferentes sistemas operativos para dispositivos móviles.

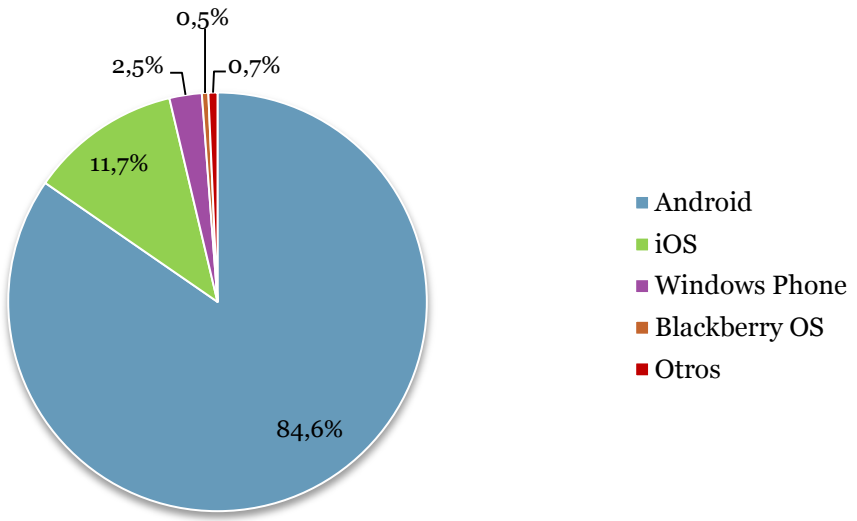



**Figura 3 - Cuota de mercado de SO para dispositivos móviles**

La aplicación ha sido desarrollada para aquellos dispositivos móviles que tengan como mínimo instalada la versión Gingerbread 2.3.

Android ha sido desarrollado con el propósito de actuar como cliente y no como servidor. Por esta razón no se puede implementar un servidor REST en Android y que sea accesible desde una red externa; sólo se podría desarrollar un servidor a nivel local.

Inicialmente se planteó que el servidor de la *smarthome* realizara las peticiones de la geolocalización al servidor implementado en Android, pero ante la imposibilidad de crear un servidor por lo mencionado en el párrafo anterior, se ha decidido que la aplicación cliente envíe cada cierto tiempo la posición al servidor.

Como matiz de lo anteriormente mencionado, existe la posibilidad de crear un servidor en Android pero solo podrá recibir peticiones de aquellas máquinas que estén en la misma red y solo en un el puerto 4444. Esto no es viable para el problema a abordar, ya que el usuario normalmente no va a estar conectado a la misma red en la que se encuentra el servidor de la vivienda inteligente.

## 2.6. Google Maps Android API v2

El propósito principal de la aplicación en Android a desarrollar es que envíe la posición al servidor, pero para proporcionar al usuario un feedback de su posición actual con respecto a la de la *smarthome*, se ha creído conveniente insertar un mapa en la aplicación con dicha información.

El API de Google Maps para Android proporciona una forma fácil, sencilla y rápida de añadir un mapa a una aplicación móvil. Dicho mapa se puede personalizar según lo que se quiera mostrar al usuario.

El uso de esta API es gratuito y permite realizar 2,500 peticiones al día. El único requisito es obtener la clave de desarrollador, mediante el registro de la aplicación en Google APIs Console. La limitación de peticiones que se pueden realizar al día no es ningún inconveniente para el desarrollo de este trabajo.

## 2.7. Google Places API

Para poder mostrar al usuario los sitios cercanos donde puede realizar la lista de la compra, se ha optado por la utilización del API de Google Places. Al igual que en el API mencionado en el subapartado anterior, la utilización de Google Places es gratuita, previa adquisición de una clave, y está limitada a 1,000 peticiones al día.

Esta API permite realizar peticiones HTTP con, entre otros parámetros, el formato de la respuesta, la posición, el radio de búsqueda y el tipo sitios. La respuesta que recibirá el cliente será en formato JSON o XML según lo haya indicado en los parámetros de la petición. Una vez tratada la respuesta, se puede mostrar en el mapa utilizando el API de Google Maps para Android.

## 2.8. XAMPP

XAMPP, acrónimo de *X* (para cualquier sistema operativo), *Apache*, *MySQL*, *PHP*, *Perl*, es una distribución de Apache gratuita y multiplataforma que contiene:

- **Apache:** servidor web HTTP de código abierto y multiplataforma que destaca por la seguridad y el rendimiento. Es el servidor web más usado hasta la fecha con una cuota del 47.92%[1]; esto se debe en parte a que

[1] Netcraft (2014), *Web Server Survey*. Disponible en: http://news.netcraft.com/archives/2014/08/27/august-2014-web-server-survey.html

permite emplear diversos lenguajes en el lado del servidor como Perl, PHP o Python, entre otros.

- **MySQL:** sistema de gestión de bases de datos (SGBD) relacional multiplataforma gratuito.
- **PHP:** lenguaje de programación diseñado para el desarrollo de sitios web con contenido dinámico.
- **Perl:** lenguaje de programación interpretado de libre uso y extensible a partir de cualquier lenguaje.

Este servidor de software libre permite al usuario una rápida instalación, ya que solo es necesario obtener y ejecutar el *.exe* (en el caso de Windows) de la web oficial (http://www.apachefriends.org). Una vez instalado, el panel de control de XAMPP permite iniciar los diferentes servicios que ofrece de una forma sencilla. A continuación se muestra una captura de mencionado panel.

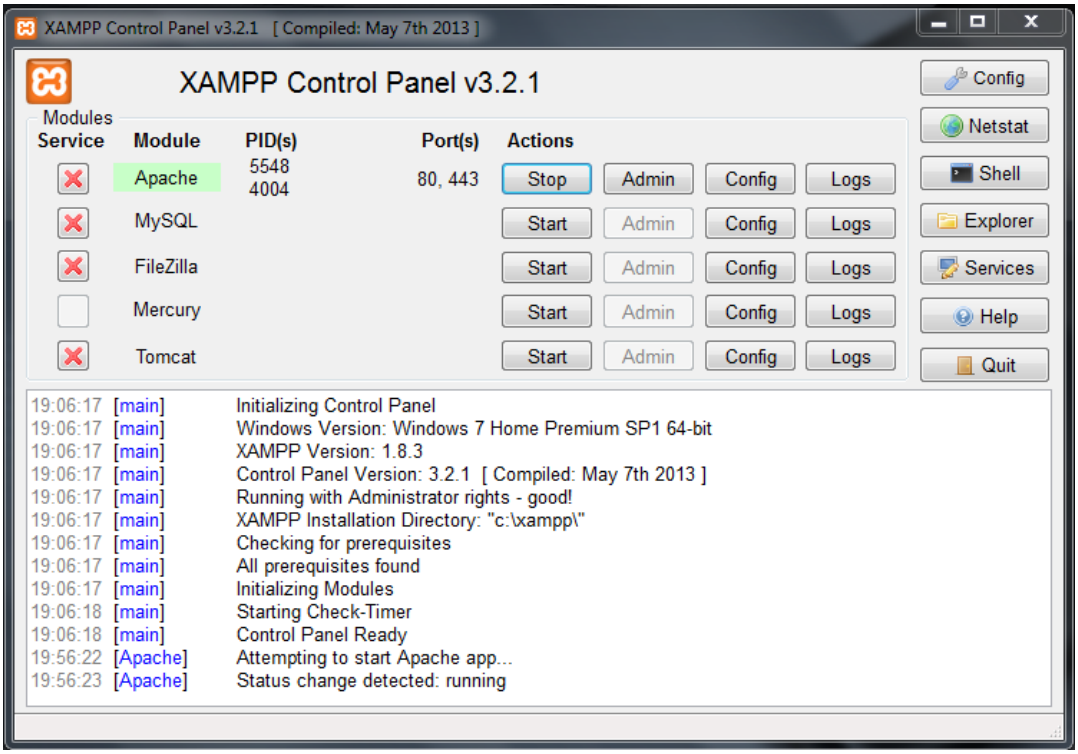



**Figura 4 - Panel de control de XAMPP**

El uso de un servidor web es necesario para poder realizar llamadas HTTP desde la aplicación móvil al servicio web que se implemente.

## 2.9. No-IP – Dynamic DNS

No-IP es un proveedor de DDNS, *Dynamic Domain Name System*, que permite la asignación de un nombre de dominio a una máquina o dispositivo con una IP dinámica o estática. No-IP ofrece un servicio gratuito de hasta tres hosts registrados con el nombre y el dominio que prefiera el usuario.

Este servicio se utiliza para poder acceder desde el exterior a una máquina personal que tenga en ejecución un servidor web o un servidor FTP, entre otros. De este modo, no es necesario conocer la IP de la máquina para acceder a ella.

Para mantener sincronizada la IP de la máquina con el nombre del dominio creado, No-IP pone a servicio del usuario la aplicación DUC, *Dynamic Update Client*, que solo hay que instalar y ejecutar en la máquina, indicando el usuario y el host a sincronizar.

El nombre de dominio que se ha creado para este trabajo es locamo.no-ip.org.

Una vez estudiado el marco tecnológico de este trabajo, se procederá a introducir el caso de estudio que se analizará, diseñará e implementará en los siguientes capítulos.

# 3. Caso de estudio

En este apartado se va a abordar la descripción de un posible escenario que puede darse dentro del contexto que engloba este trabajo.

Juan es un apasionado de las nuevas tecnologías y hace unos meses que ha instalado en su casa un sistema domótico para hacer de su hogar una "vivienda inteligente". La empresa proveedora de este servicio le ha recomendado que instale la nueva aplicación que ha sacado al mercado para que el usuario no tenga que preocuparse por apagar las luces o la calefacción cuando se va de casa ya que, según su posición, la casa "sabe" qué hacer. También le han indicado que puede añadir elementos a la lista de la compra (en caso de necesitar algo) y que el resto de los habitantes de la vivienda podrán ver en su *smartphone* qué hay que comprar.

A Juan le entusiasma la idea, así que se dispone a probarla. Para ello, lo primero que hace es instalarse la aplicación que le han proporcionado en su *smartphone* con Android KitKat; además activa el GPS para que se pueda mandar su posición a la vivienda. Una vez se ha instalado, se abre automáticamente y lo que tiene que hacer es iniciarla. Antes de poder enviar su localización a su *smarthome*, debe registrarse con el nombre de usuario que él prefiera y la contraseña que le han indicado los de la compañía.

Una vez registrado, lo primero que le aparece es un mapa con la posición de su casa y su posición actual. Ahora mismo está cerca de casa, así que decide ir a dar una vuelta por la ciudad. Dando el paseo recuerda que le habían dicho los de la empresa de domótica que podía introducir lo que necesitaba de compra en la aplicación, así que accede a la "Lista de la compra" para añadirle el pollo y los huevos que necesita para la cena.

Continúa andando y recibe una notificación en su móvil de la aplicación indicándole que tiene supermercados cerca donde puede comprar los elementos que había introducido anteriormente. Al pulsar sobre la notificación, se muestra en el mapa los sitios donde puede comprar y al pulsar encima de uno de ellos, le aparece el nombre del supermercado. Al salir de la tienda con los productos, Juan abre la lista de la compra otra vez para borrar lo que ha comprado; para ello, mantiene pulsado cada elemento de la lista durante un segundo para poder eliminarlo. Haciendo esto, el resto de dispositivos se sincronizarán y el resto de su familia podrá ver que ya no hay que comprar nada.

Ahora Juan puede estar tranquilo porque confía en que su casa sabrá qué hacer dependiendo de dónde se encuentre en cada momento: si se está alejando, por ejemplo se apagarán las luces y/o la calefacción o el aire acondicionado (dependiendo de la época del año); y se encenderán las luces auxiliares y la calefacción o el aire acondicionado si está regresando a casa.

Definido el caso de estudio, se procederá en el siguiente capítulo a analizar el problema que ha planteado, especificando los requisitos que debe cumplir, el modelo conceptual y la interfaz gráfica de usuario.

# 4. Análisis del problema

En este capítulo se plantea el análisis del caso de estudio descrito en el apartado anterior con el fin de describir los requisitos y funcionalidades de la aplicación móvil y el servidor para su posterior desarrollo.

## 4.1. Descripción del problema

El problema que se ha planteado en el capítulo anterior implica desarrollar una aplicación móvil que envíe la posición del dispositivo cada cierto tiempo a un servidor. El servidor, del que se hará referencia en este documento como TFGLocation a partir de este momento, analizará las posiciones que reciba y aplicará ciertas reglas sobre la casa, tales como encender luces o apagar la calefacción. Además, este servidor tendrá la lista de la compra actualizada con los elementos que el usuario vaya añadiendo y borrando; el resto de habitantes de la casa, también tendrán la lista actualizada en sus dispositivos.

El desarrollo de este proyecto implica su integración con un proyecto de previo desarrollo denominado SmartHome, que será el servidor que reciba y lleve a cabo las acciones que hay que realizar sobre la casa.

SmartHome es un servidor HTTP que proporciona una capa de servicios REST para interactuar con las funcionalidades (*Device Funcionalities*) de una vivienda inteligente. Se entiende como "funcionalidades" las luces, las persianas, la climatización, los enchufes, los electrodomésticos, los diversos sensores (de lluvia, de viento, de temperatura, de movimiento, etc.), es decir, cualquier dispositivo sobre el que se pueda realizar una acción. Las únicas operaciones permitidas sobre estas funcionalidades son los métodos GET, que obtiene información del recurso, y PUT, que solicitará la ejecución de una acción sobre el recurso, indicando dicha acción en el payload de la petición.

## 4.2. Especificación de requisitos

La especificación de las funcionalidades y requisitos que deben cumplir tanto el servidor como la aplicación cliente constituyen la base del diseño e implementación. Las características que debe tener el servidor no son las mismas que las del cliente; por ello, y en pos de una mayor claridad, se diferenciarán sus requisitos.

### 4.2.1. Especificación de requisitos del servidor

A continuación se detallan los requisitos y las funcionalidades que tiene que cumplir el servidor TFGLocation:

- Permitirá el registro de nuevos usuarios.
- Permitirá la introducción de la nueva localización de un usuario y obtener una lista con todas sus posiciones.
- Deberá definir unas reglas de comportamiento según la geoposición del usuario con respecto de la casa y enviar las acciones sobre ciertas funcionalidades que de ellas deriven al servidor SmartHome.
- Permitirá añadir y eliminar elementos a la lista de la compra y que ésta se sincronice con todos los dispositivos conectados.

### 4.2.2. Especificación de requisitos de la aplicación cliente

Una vez analizados con requisitos del servidor, hay que definir los del cliente:

- La aplicación se diseñará de forma que el usuario, tanto si domina las nuevas tecnologías como si no está familiarizado con ellas, pueda interactuar con la aplicación de una forma sencilla e intuitiva sin tener que realizar muchas acciones sobre ella.
- Podrá crearse un nuevo usuario o registrarse uno ya existente. No será necesario que el usuario se registre cada vez que se inicie la aplicación. Existirá la posibilidad de que el usuario cierre su sesión.
- Se enviará la geolocalización al servidor TFGLocation; dicho envío será completamente transparente al usuario.
- Aparecerá un mapa con la posición de la casa y la posición actual del usuario, distinguiéndose ambos por el uso de diferentes iconos.
- Podrán añadirse y eliminarse elementos de la lista de la compra; la sincronización con el servidor es transparente al usuario.
- Deberá notificar que hay supermercados cercanos al usuario y al ver dicha notificación deberá mostrarlos en el mapa junto con su nombre. Esto solo ocurrirá si hay elementos en la lista de la compra.
- La interfaz deberá ser sencilla, intuitiva y fácil de usar.

## 4.3. Modelo conceptual

Entendiendo modelo como una simplificación de la realidad, se va a construir un modelo para comprender mejor la capa lógica del servidor y de la aplicación que se va a implementar. De esta forma se consigue visualizar cómo va a ser el sistema y especificar su estructura y comportamiento.

Para tener una visión más clara de la estructura del servidor se va a hacer uso de un diagrama de clases. En éste se van a reflejar las diferentes clases que forman la lógica y la relación entre ellas incluyendo, en cada una de estas clases, los atributos y las operaciones que se pueden realizar sobre cada objeto. En las operaciones de cada clase se han omitido los métodos getters y setters de los atributos por simplificar el modelo, aunque sí se implementarán.

A continuación se muestra el diagrama de clases en UML[2]. Para su elaboración se ha utilizado StarUML, herramienta ampliamente utilizada por empresas de la talla de Samsung, Intel o Accenture.

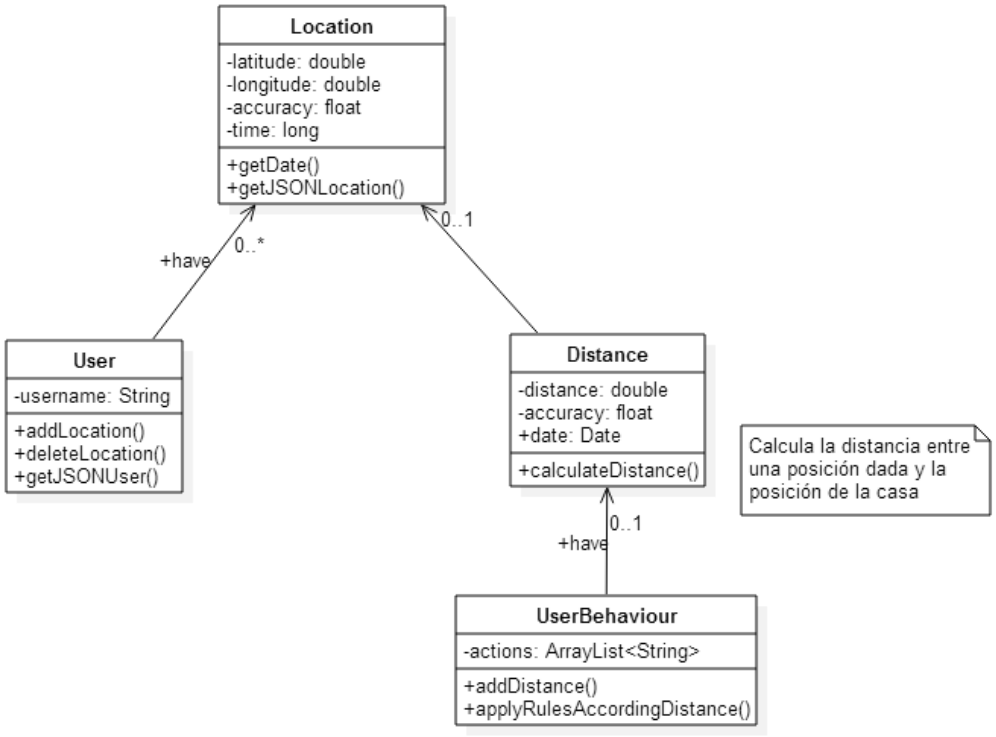



**Figura 5 - Diagrama de clases del servidor**

[2] UML: *Unified Modeling Language*. Es un lenguaje gráfico de modelado de software que permite visualizar especificar construir y documentar un sistema.

## 4.4. Interfaz de usuario

La forma en la que el usuario interactúa con el sistema es crucial, razón por la que se debe centrar el desarrollo de la interfaz pensando siempre en el usuario final y haciendo que la aplicación sea usable. Se entiende el concepto de usabilidad como aquel atributo de una aplicación relacionado con la facilidad de uso, la rapidez de aprendizaje del usuario y la eficiencia al utilizarla.

Siguiendo una de las frases más memorables de Leonardo da Vinci de "La simplicidad es la máxima sofisticación", se ha intentado hacer sencilla la forma en la que el usuario interactúa con la aplicación.

Una de las opciones más sencillas para poder realizar los *mockups* o bocetos es la herramienta Pencil, que proporciona al usuario elementos gráficos para la elaboración de los prototipos de las interfaces de la forma más real posible. A continuación se muestran los diferentes bocetos que se han diseñado.


Esta será la pantalla que el usuario visualizará cuando abra la aplicación y tendrá dos opciones: iniciarla o salir de ella.

**Figura 6 - Boceto de la pantalla inicial de la aplicación**

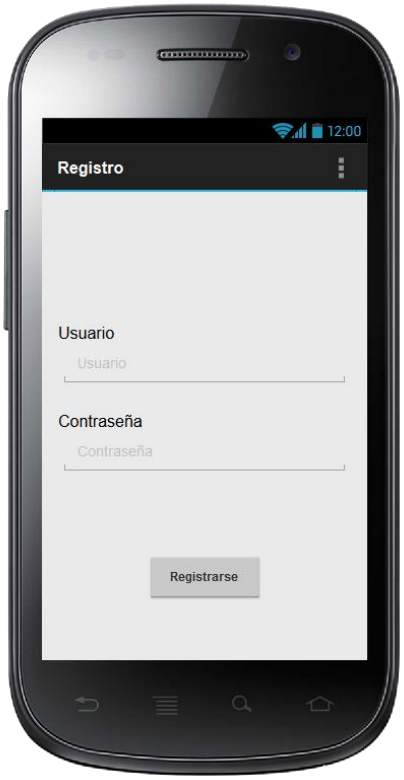



Figura 7 - Boceto de la pantalla de registro

El registro es obligatorio cuando se instala la aplicación o cuando se ha cerrado sesión anteriormente y se quiere volver a acceder.

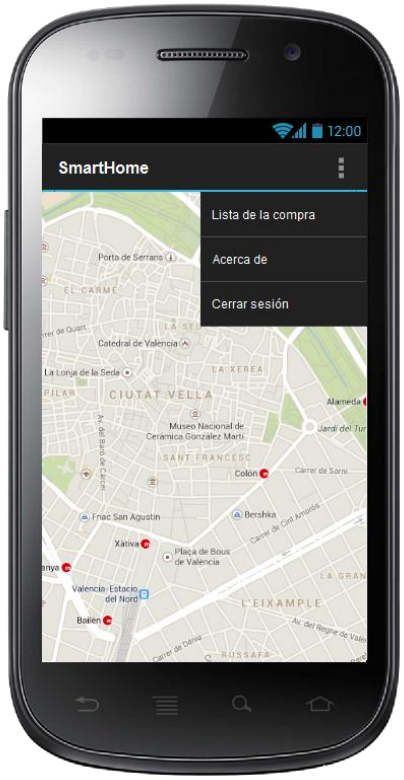
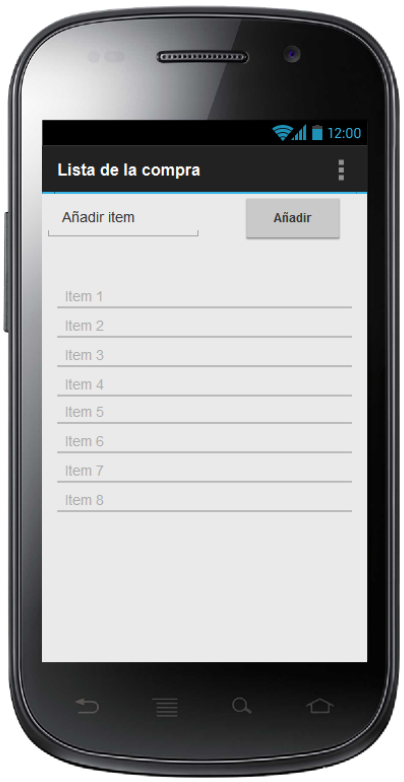



Figura 8 - Boceto de la pantalla principal

Ésta será la pantalla principal de la aplicación y mostrará la posición de la vivienda y la posición actual del usuario. Si hay elementos en la lista de la compra, mostrará los supermercados más cercanos.

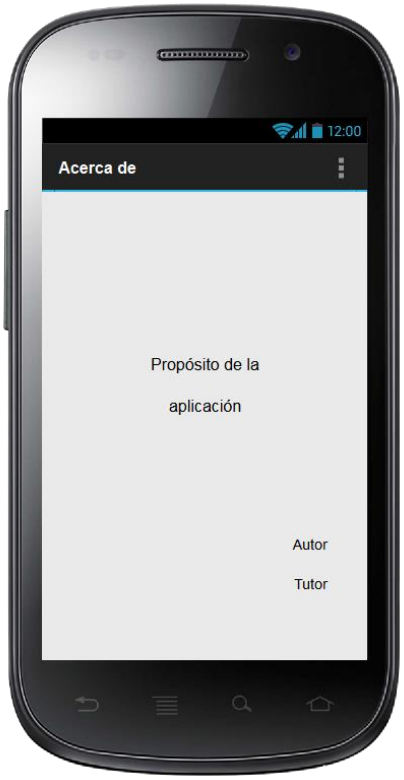
El usuario podrá escoger del menú de opciones entre ver la lista de la compra, obtener información de la aplicación o cerrar su sesión.




El usuario podrá añadir o borrar elementos de la lista de la compra en esta pantalla.

Para añadir un elemento, escribirá el nombre y pulsará "Añadir". Para borrarlo, tendrá que pulsar durante un segundo el elemento que desee eliminar.

**Figura 9 - Boceto de la gestión de la lista de la compra**




Esta pantalla mostrará información acerca de la aplicación y el autor y tutor de este trabajo.

**Figura 10 - Boceto del *Acerca de* de la aplicación**

En este capítulo se ha estudiado con detalle los requisitos que se deben tener en cuenta a la hora de diseñar tanto la aplicación cliente como el servidor. El capítulo siguiente aportará soluciones al problema que se ha planteado en el caso de estudio ciñéndose a lo descrito a lo largo de esta sección.

# 5. Diseño de la solución

Una vez analizado el problema planteado en el caso de estudio, se procede a la descripción en este capítulo de qué soluciones se aportarán con el fin de desarrollar tanto el servidor como la aplicación móvil.

## 5.1. Propuesta de solución al problema planteado

La idea inicial para que el servidor aplicara unas normas de comportamiento era que enviara al cliente una petición para conocer su posición y éste respondiera, evitando así carga al smartphone. Tal y como se ha visto en el apartado 2.4, esta idea no ha podido llevarse a cabo; una alternativa es que la aplicación envíe su posición al servidor a razón de intervalos de tiempo previamente definidos y, de esta forma, éste ya puede aplicar las reglas definidas.

La comunicación entre los distintos componentes, tanto las respuestas del servidor SmartHome como los payloads que se le envían, están implementadas de forma que se devuelva en formato JSON. Es debido a ello por lo que la comunicación entre el servidor que se va a implementar, el denominado como TFGLocation, como el ya existente va a ser utilizando el formato JSON para las peticiones y respuestas: ambos tendrán una capa de servicios REST. Siguiendo esta línea, es lógico que la aplicación móvil cliente también utilice este servicio para la comunicación con el nuevo servidor.

La acotación del tiempo a dedicar a este trabajo supone que sólo se hayan podido desarrollar los prototipos tanto del servidor como de la aplicación cliente. Ambas partes admiten mejoras, aunque la más destacable es la del servidor porque sólo va a aplicar las acciones definidas en las reglas de comportamiento en base a un solo usuario.

## 5.2. Servidor web basado en REST

El servidor TFGLocation es un servidor HTTP con una capa REST que se va a encargar de recibir las peticiones del cliente y va a procesarlas para definir unas reglas que realizarán ciertas acciones sobre las funcionalidades de la casa.

A continuación se explicará el funcionamiento general de las clases definidas en diagrama de clases del capítulo anterior y algunas clases auxiliares que se utilizarán.

### 5.2.1. User

Esta clase contiene el nombre de usuario y una lista con las posiciones (*locations*) de cada usuario que se registre en la aplicación móvil. Cada vez que se invoque el método PUT de un determinado usuario, se actualizará la lista de posiciones añadiendo la nueva localización.

Contiene un método (`getJSONUser()`) que creará un objeto JSON de un determinado usuario con su nombre y la lista actualizada de sus posiciones. Este último método será llamado cuando se invoque el método GET de un usuario concreto.

### 5.2.2. Location

La clase `Location` contiene los atributos típicos de una geolocalización: longitud, latitud, precisión (que viene dada por la precisión obtenida por el proveedor del servicio de localización) y la fecha, en milisegundos, en la que obtuvo.

Contiene un método que creará un objeto JSON con los atributos mencionados y que será invocado por el método `getJSONUser()` para crear un array JSON con las posiciones del usuario.

### 5.2.3. Distance

`Distance` es la clase en la que se calcula la distancia de la posición enviada por la aplicación móvil y la posición de la vivienda. Además contiene la precisión de esa distancia y la fecha y hora en la que obtuvo.

Para el cálculo de la distancia entre una posición dada y la de la casa, se ha utilizado la fórmula del Haversine, que calcula la distancia de círculo máximo entre dos puntos de un globo sabiendo la longitud y latitud de cada uno de ellos.

$$haversin\left(\frac{d}{R}\right) = haversin(\varphi_1 - \varphi_2) + \cos(\varphi_1)\cos(\varphi_2)\, haversin(\Delta\lambda)$$

donde $haversin$ es $haversin(\theta) = sin^2\left(\frac{\theta}{2}\right) = (1 - \cos(\theta))/2$,
$d$ es la distancia entre dos puntos,
$R$ es el radio de la esfera,
$\varphi_1$ es la latitud del punto 1,
$\varphi_2$ es la latitud del punto 2, y
$\Delta\lambda$ es la diferencia de longitud.

### 5.2.4. UserBehaviour

En esta clase se almacena una lista con todas las distancias (`Distance`) calculadas y en la que se comprueban las cuatro posiciones anteriores a la última posición insertada y se aplican unas reglas (definidas en la clase Rules) según si el usuario se está alejando o acercando y la distancia calculada.

A continuación se va a esquematizar el estudio del comportamiento del usuario:

- El usuario se está alejando de la casa si la mayoría de las últimas cuatro distancias calculadas son menores que la última recibida. Si esto es así, se mira la última posición recibida:
    - Si está entre 200 y 500 metros de la casa, se aplica la regla `twoHundredMetersMovingAwayRule()`.
    - Si está entre 500 metros y 1 kilómetro de la casa, se aplica la regla `fiveHundredMetersMovingAwayRule()`.
    - Si está a 1 kilómetro o más de la casa, se aplica la regla `oneThousandMetersMovingAwayRule()`.
- El usuario se está acercando a casa si la mayoría de las últimas cuatro distancias calculadas son mayores que la última recibida. Se aplican las siguientes reglas dependiendo de la última posición:
    - Si está entre 200 y 500 metros de la casa, se aplica la regla `twoHundredMetersApproachingRule()`.
    - Si está entre 500 metros y 1 kilómetro de la casa, se aplica la regla `fiveHundredMetersApproachingRule()`.
    - Si está a 1 kilómetro o más de la casa, se aplica la regla `oneThousandMetersApproachingRule()`.

La siguiente imagen ilustra de una forma gráfica las reglas que se aplicarían según la posición del usuario como se acaba de explicar.

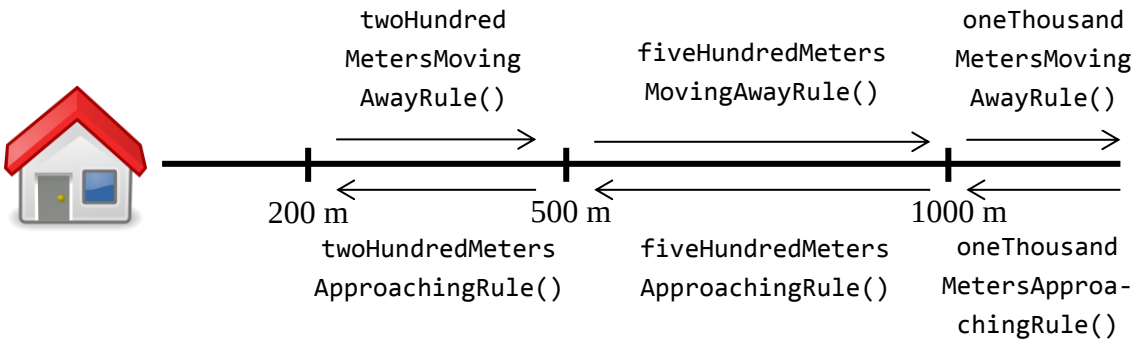



**Figura 11 - Aplicación de reglas según la posición del usuario**

Además de aplicar las reglas, se tendrá una lista con la última regla aplicada; de este modo, no se aplicará una regla si la última regla y ésta coinciden. Esto también sirve para que, en caso de que el usuario inicie la aplicación cuando esté lejos de casa o no se hayan podido enviar sus últimas posiciones, se aplique la regla correspondiente la posición actual y las reglas de las distancias menores o mayores, dependiendo de si se está alejando o acercando respectivamente.

Para poder entender mejor la descripción anterior, se va a poner un ejemplo de un escenario concreto: el usuario inicia la aplicación y se encuentra a 1,5 kilómetros de casa; entonces, se comprobará que anteriormente no se había aplicado la regla de los 1000 metros ni tampoco la de los 500 y 200 metros, así que aplicará las tres. Otro escenario podría ser el de que el usuario ha perdido la conexión durante un tiempo y cuando la recupera, le envía la posición al servidor, que recibe que está a, por ejemplo, 300 metros de la casa; en este caso se seguiría el mismo procedimiento de antes y se aplicaría la regla de los 200 metros y también la de los 500 metros y la de los 1000 metros.

Por último, si la posición del usuario no varía con respecto a las anteriores, indica que está mucho tiempo en el mismo sitio y por lo tanto, es conveniente seguir el mismo procedimiento que el explicado anteriormente (aplicando las reglas de cuando el usuario se está alejando).

### 5.2.5. Rules

Una vez explicado el comportamiento del usuario y dependiendo de su posición actual y las anteriores, se van a aplicar unas reglas que no son más que acciones sobre ciertas funcionalidades de la smarthome. Estas acciones se envían al servidor SmartHome.

A continuación se van a detallar cada una de las reglas citadas en el subapartado anterior:

- `twoHundredMetersMovingAwayRule()`:
    - Se apagan todas las luces centrales de las habitaciones y se dejan encendidas solo las luces auxiliares de la cocina y el comedor.
    - Se apagan los calefactores de los baños.
    - Se activa el sensor de movimiento y el sensor que comprueba que la puerta principal está cerrada.
- `fiveHundredMetersMovingAwayRule()`: si estamos en los meses de entre mayo y octubre inclusive:
    - Se bajan cuatro de las cinco persianas del comedor y las de las habitaciones.
    - Si es de día, se abren una persiana del comedor y los estores de la habitación principal y la de los niños.
    - Si es de noche, se bajan también la persiana que quedaba por bajar del comedor y los estores de las habitaciones principal e infantil.
- `oneThousandMetersMovingAwayRule()`: dependiendo de la época del año, se desactivarán ciertos dispositivos de climatización:
    - Si es verano, se desactivará el aire acondicionado del comedor, de la habitación principal y de la infantil.
    - Si es invierno, se desactivará el climatizador del comedor y los calefactores de las tres habitaciones.
- `twoHundredMetersApproachingRule()`: se encenderán las luces auxiliares del comedor y de la cocina.
- `fiveHundredMetersApproachingRule()`: se abrirán todas las persianas del comedor, los estores de la habitación principal y de la infantil, así como la persiana del estudio solo en los meses de noviembre a abril y si es de día.

- `oneThousandMetersApproachingRule()`: dependiendo de la época del año, se encenderá la calefacción o el aire acondicionado:
    - Si es verano, se activará l aire acondicionado del comedor, de la habitación de matrimonio y la de los niños.
    - Si es invierno, se encenderá la calefacción del comedor y la de las tres habitaciones (matrimonio, infantil y estudio).

## 5.3. Aplicación móvil

La aplicación móvil será la encargada de enviar la posición actual del dispositivo al servidor TFGLocation cada cierto tiempo y de forma transparente al usuario y además, también servirá para que el usuario pueda gestionar la lista de la compra.

En los siguientes subapartados se van a describir con más detalle cada uno de los bocetos diseñados en el apartado 4.4.

### 5.3.1. Pantalla inicial

La pantalla inicial será la pantalla que se muestre al usuario cada vez que se abra la aplicación. Se podrán realizar dos acciones: iniciar la aplicación o salir de ella.

Si el usuario decide iniciar la aplicación, el comportamiento será el siguiente: si el usuario no se ha registrado antes en la aplicación o, si lo estaba, ha cerrado sesión, tendrá que iniciar sesión. Si el usuario ya ha iniciado sesión, le llevará a la pantalla principal.

### 5.3.2. Pantalla de inicio de sesión

La pantalla de inicio de sesión o de registro tendrá la función de iniciar sesión, en caso de que el usuario ya se haya registrado anteriormente contra el servidor, o de registrarse. Para ambos casos, tendrá que ingresar un nombre de usuario y la contraseña que estará definida en el servidor; si no se introduce correctamente la contraseña, no podrá iniciar la aplicación.

En caso de se hayan introducido todos los datos correctamente, al darle al botón de “Registro”, se pasará a la pantalla principal.

### 5.3.3. Pantalla principal

Una vez iniciada sesión, en la pantalla inicial se mostrará un mapa con la posición en la que se encuentra la casa y, al cabo de un tiempo, la posición actual del usuario.

Se mostrará un menú en el que el usuario podrá seleccionar entre acceder a la lista de la compra, obtener información acerca de la aplicación o cerrar sesión. Si accede a "Lista de la compra", le llevará a la pantalla correspondiente (explicada en el siguiente subapartado). La opción de "Acerca de" le llevará a la pantalla de acerca de. En caso de haber elegido "Cerrar sesión", la sesión actual se cerrará, con lo que no se enviarán la localizaciones al servidor; y le llevará a la pantalla inicial de nuevo.

Si hay objetos en la lista de la compra, se añadirán al mapa los supermercados más cercanos con respecto la posición del usuario.

### 5.3.4. Pantalla de la lista de la compra

Si el usuario decide acceder a la lista de la compra, que es una opción del menú de la pantalla principal, podrá gestionarla, es decir, añadir o borrar elementos. Se mostrará una lista con los diferentes ítems en caso de que él o algún habitante de la casa hayan añadido elementos.

Para poder añadir elementos, el usuario introducirá el nombre del producto que necesita comprar y pulsará el botón "Añadir". Si por el contrario desea eliminarlos, tendrá que pulsar durante un segundo cada uno de los ítems a borrar.

En todo momento, la lista estará actualizada en el servidor ya que se enviará una petición por cada una de las acciones descritas en el párrafo anterior. Esto será transparente al usuario.

### 5.3.5. Pantalla de "acerca de"

Esta pantalla mostrará información acerca de la aplicación: mostrará el propósito de la misma y el desarrollador y el revisor de la aplicación, que serán el autor y el tutor de este trabajo respectivamente.

## 5.4. Arquitectura cliente-servidor

La arquitectura en la que se basará este proyecto es en la arquitectura cliente-servidor, en el que la aplicación móvil actuará únicamente como cliente y el servidor TFGLocation actuará tanto como servidor como cliente del servidor SmartHome.

La interacción entre el cliente y el servidor va realizará mediante peticiones HTTP utilizando los métodos GET, POST y DELETE. Todos los recursos del servidor van a tener los dos primeros métodos implementados; el método DELETE solo implementará para eliminar elementos de la lista de la compra.

En la siguiente Figura se muestra la interacción entre los distintos clientes y servidores:

- **Aplicación móvil:** enviará peticiones al servidor TFGLocation con la posición y las acciones sobre la lista de la compra.
- **Servidor TFGLocation:** recibirá las peticiones del cliente móvil y enviará peticiones al servidor SmartHome con las acciones a realizar sobre las funcionalidades de la vivienda según los datos que reciba de la aplicación móvil.
- **Servidor SmartHome:** recibirá las peticiones de TFGLocation y realizará las acciones que le haya enviado.

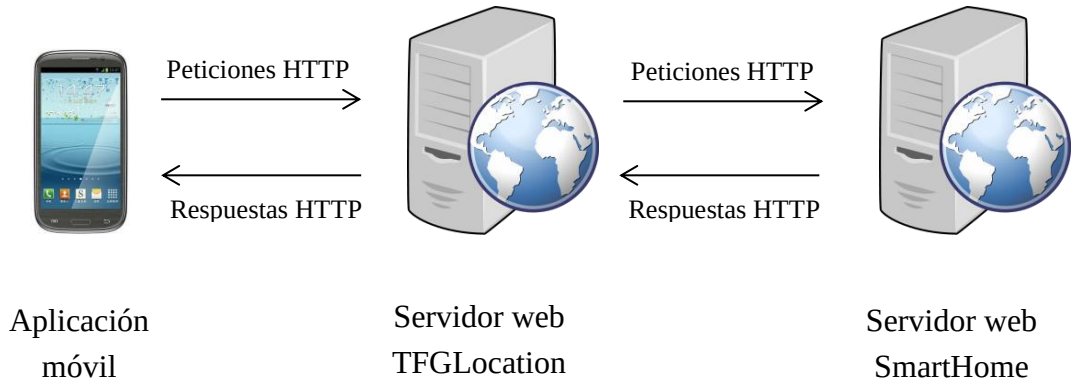



**Figura 12 - Arquitectura cliente-servidor del proyecto**

En esta sección se ha estudiado cómo solucionar el problema planteado en el caso de estudio especificando la funcionalidad que debe tener cada una de las clases de la capa lógica del servidor y de la aplicación. La implementación de los prototipos se explica en el capítulo siguiente.

# 6. Implementación

Una vez expuestas las etapas de análisis y diseño, en este capítulo se expondrá la parte de la implementación del servidor HTTP basado en REST y la correspondiente a la aplicación móvil.

Si el lector de este trabajo lo desea, puede ver el código fuente de ambos proyectos en el siguiente enlace: https://github.com/locamo/tfg.

## 6.1. Servidor HTTP basado en Restlet

El servidor HTTP se ha desarrollado utilizando el framework Restlet, que proporciona al programador sencillez a la hora de implementarlo, tal y como se ha expuesto en el apartado 3.2.

La estructura de este apartado se va a dividir entre la implementación de las distintas clases que conforman la capa lógica (incluyendo las reglas que se aplicarán según el comportamiento del usuario), la creación del servidor y la creación de los recursos. Para tener una idea más clara, a continuación se muestra la estructura de clases y paquetes del servidor y las librerías utilizadas.

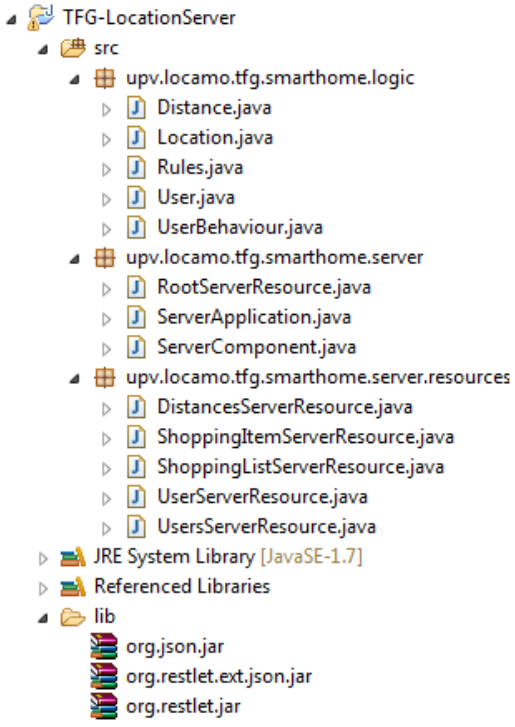



**Figura 13 - Estructura del servidor TFGLocation**

### 6.1.1. Clases que conforman la capa lógica

A continuación se muestra la implementación en Java de la capa lógica del servidor, que la conforman las clases descritas en la etapa de diseño. No se describe la funcionalidad de las mismas porque ya se ha explicado en el capítulo 6, concretamente en el apartado 6.2. Los recursos del servidor, que me mostrarán en el subapartado 7.1.2., harán uso de estas clases.

Para una mayor simplicidad y claridad, se van a mostrar solo aquellos métodos más relevantes de cada clase y se comenta en el mismo código su funcionalidad.

#### 6.1.1.1. User

El código fuente en Java para la clase `User` es el que sigue:

```
package upv.locamo.tfg.smarthome.logic;

import java.util.ArrayList;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

public class User {

	private String username = "";
	private ArrayList<Location> locationList = null;

	/**
	 * Constructor
	 * @param uname
	 * @param locationList
	 */
	public User(String uname, ArrayList<Location> locationList) {
		this.username = uname;
		this.locationList = locationList;
	}

	[…]

	/**
	 * This method creates a JSONObject with the name of the user and his location
	 * @return JSONObject with name and a JSONArray with positions
	 * @throws JSONException
	 */
	public JSONObject getJSONUser() throws JSONException {
		JSONObject jsonResult = new JSONObject();
		JSONArray jsonArray = new JSONArray();
		jsonResult.put("userID", username);
		if (locationList != null)
			for (Location l : locationList)
				jsonArray.put(l.getJSONPosition());
		jsonResult.put("location", jsonArray);
		return jsonResult;
	}
}
```

**Figura 14 - Código fuente de la clase `User`**

El siguiente ejemplo muestra cómo quedaría la representación de un usuario en formato JSON: el nombre de usuario sería "locamo" y se tendrían registradas dos localizaciones.

```
{
   "userID": "locamo",
    "location":
    [
       {
         "date": "Tue Aug 12 21:23:39 CEST 2014",
         "latitude": 38.7181938,
         "accuracy": 21,
         "longitude": -0.6662247
       },
       {
         "date": "Tue Aug 12 21:41:10 CEST 2014",
         "latitude": 38.71713323,
         "accuracy": 53,
         "longitude": -0.65893525
       }
    ]
}
```

**Figura 15 - Ejemplo de representación de un usuario en JSON**

#### 6.1.1.2. Location

La implementación de la clase `Location` queda de la siguiente forma:

```
package upv.locamo.tfg.smarthome.logic;

import java.util.Date;

import org.json.JSONException;
import org.json.JSONObject;

public class Location {

	private double longitude;
	private double latitude;
	private float accuracy;
	private long time;

	/**
	 * Constructor
	 * @param longitude
	 * @param latitude
	 * @param time
	 * @param accuracy
	 */
```

```
	public Location(double longitude, double latitude, long time, float accuracy){
		this.longitude = longitude;
		this.latitude = latitude;
		this.time = time;
		this.accuracy = accuracy;
	}

	/**
	 * Constructor
	 * @param longitude
	 * @param latitude
	 */
	public Location(double longitude, double latitude){
		this.longitude = longitude;
		this.latitude = latitude;
	}

	/*
	 * Getters for Location attributes
	 */
	[…]

	public Date getDate() {
		Date d = new Date(time);
		return d;
	}

	/**
	 * Creates a JSON object of Location
	 * @return JSONObject
	 */
	public JSONObject getJSONPosition(){
	 JSONObject jsonObjLocation = new JSONObject();
	 try {
		jsonObjLocation.put("longitude", longitude);
			jsonObjLocation.put("latitude", latitude);
			jsonObjLocation.put("accuracy", accuracy);
		 jsonObjLocation.put("date", getDate());
		 return jsonObjLocation;
		} catch (JSONException e) {
			e.printStackTrace();
			return null;
		}
	}

}
```

**Figura 16 - Código fuente de la clase `Location`**

La representación de un objeto de tipo `Location` sería la siguiente:

```
{
  "date": "Tue Aug 12 21:23:39 CEST 2014",
  "latitude": 38.7181938,
  "accuracy": 21,
  "longitude": -0.6662247
}
```

**Figura 17 - Ejemplo de representación de una localización en JSON**

#### 6.1.1.3. Distance

A continuación se muestra la implementación de la clase `Distance`:

```
package upv.locamo.tfg.smarthome.logic;

import java.util.Date;

public class Distance {

	// Location of home
	public static Location home = new Location(-0.6591402085289246,
			38.71733000907843);

	private double distance;
	private float accuracy;
	private Date date;

	/**
	 * Constructor
	 * @param distance
	 * @param accuracy
	 * @param date
	 */
	public Distance(double distance, float accuracy, Date date) {
		this.distance = distance;
		this.accuracy = accuracy;
		this.date = date;
	}

	/*
	 * Getters for Distance attributes
	 */
	[…]

	/**
	 * Haversine formula to calculate distance between a location and home
	 * @param longitude
	 * @param latitude
	 * @return distance (in meters)
	 */
	public static double calculateDistance(Location loc) {
		double earthRadius = 6371000; // meters
		double dLat = Math.toRadians(loc.getLatitude() - home.getLatitude());
		double dLng = Math.toRadians(loc.getLongitude() -
home.getLongitude());
		double a = Math.sin(dLat / 2) * Math.sin(dLat / 2)
			+ Math.cos(Math.toRadians(home.getLatitude()))
			* Math.cos(Math.toRadians(loc.getLatitude()))*Math.sin(dLng/2)
			* Math.sin(dLng / 2);
		double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
		float dist = (float) (earthRadius * c);
		return dist;
	}
}
```

**Figura 18 - Código fuente de la clase `Distance`**

#### 6.1.1.4. UserBehaviour

La clase en la que se implementa el algoritmo que define qué reglas aplicar según el histórico de posiciones del usuario se muestra a continuación:

```
package upv.locamo.tfg.smarthome.logic;

import java.util.ArrayList;

public class UserBehaviour {

	private static ArrayList<Distance> distances = new ArrayList<Distance>();

	private static ArrayList<String> actions = new ArrayList<String>();

	[…]

	public static void applyRulesAccordingDistance(Distance dist) {

	   Rules rule = new Rules();

	   // This array contains this numbers:
	   // -1: distances.get(i) > distances.get(i + 1) - indicates that user
	   // is approaching
	   // 0: distances.get(i) = distances.get(i + 1) - indicates that user is
	   // at the same location
	   // 1: distances.get(i) < distances.get(i + 1) - indicates that user is
	   // moving away
	   int sizeAux = 4;
	   int[] aux = new int[sizeAux];
	   int index = distances.indexOf(dist);

	   if (distances.size() != 0 && index >= 4) {
	      for (int i = 4; i >= 0; i--) {
	         int j = index - i;
	         if (j < 4){
	            if (distances.get(index).getDistance() >
	                distances.get(j).getDistance())
	               aux[sizeAux - i] = 1;
	            else if (distances.get(index).getDistance() <
	                     distances.get(j).getDistance())
	               aux[sizeAux - i] = -1;
	            else
	               aux[sizeAux - i] = 0;
	         }
	   }
	   // User is approaching to home
	   if (countEqualNumbers(aux, -1) + countEqualNumbers(aux, 0) >
	       countEqualNumbers(aux, 1) + countEqualNumbers(aux, 0)) {

	      // If user is 200 to 500 meters home, activate
	      // "twoHundredMetersApproachingRule"
	      if (dist.getDistance() >= 200 && dist.getDistance() < 500) {
	           if ((actions.size() != 0 &&
	                !actions.get(actions.size() - 1).equals("200mApproaching"))) {

	              rule.twoHundredMetersApproachingRule();
	              actions.add("200mApproaching");
	           }
```

```
            else if (!actions.get(actions.size()-1).equals("500mApproaching")
                     || actions.size() == 0){

               rule.fiveHundredMetersApproachingRule(dist.getDate());
               actions.add("500mApproaching");
               rule.twoHundredMetersApproachingRule();
               actions.add("200mApproaching");
            }
            else if (!actions.get(actions.size()-1).equals("1000mApproaching")
                     || actions.size() == 0){

               rule.oneThousandMetersApproachingRule(dist.getDate());
               actions.add("1000mApproaching");
               rule.fiveHundredMetersApproachingRule(dist.getDate());
               actions.add("500mApproaching");
               rule.twoHundredMetersApproachingRule();
               actions.add("200mApproaching");
            }
         }

         // If user is 500 to 1000 meters home, activate
         // "fiveHundredMetersApproachingRule"
         else if (dist.getDistance() >= 500 && dist.getDistance() < 1000) {
            if ((actions.size() != 0 &&
                 !actions.get(actions.size() - 1).equals("500mApproaching"))) {

               rule.fiveHundredMetersApproachingRule(dist.getDate());
               actions.add("500mApproaching");
            }
            else if (!actions.get(actions.size()-1).equals("1000mApproaching")
                     || actions.size() == 0){

               rule.oneThousandMetersApproachingRule(dist.getDate());
               actions.add("1000mApproaching");
               rule.fiveHundredMetersApproachingRule(dist.getDate());
               actions.add("500mApproaching");
            }
         }

         // If the is 1000 or more meters home, activate
         // "oneThousandMetersApproachingRule"
         else if (dist.getDistance() >= 1000){
            if ((actions.size() != 0 &&
                 !actions.get(actions.size() -1).equals("1000mApproaching")) ||
                 actions.size() == 0) {

               rule.oneThousandMetersApproachingRule(dist.getDate());
               actions.add("1000mApproaching");
            }
         }

      }

      // User is moving away from home
      else if (countEqualNumbers(aux, -1) + countEqualNumbers(aux, 0) <
               countEqualNumbers(aux, 1) + countEqualNumbers(aux, 0)) {

         // If user is 200 to 500 meters home, activate
         // "twoHundredMetersApproachingRule"
         if (dist.getDistance()>=200 && dist.getDistance()<500) {
            if ((actions.size() != 0 &&
                 !actions.get(actions.size() - 1).equals("200mMovingAway"))
                || actions.size() == 0) {
```

```
                rule.twoHundredMetersMovingAwayRule();
                actions.add("200mMovingAway");
            }
        }

        // If user is 500 to 1000 meters home, activate
        // "fiveHundredMetersApproachingRule"
        else if (dist.getDistance() >= 500 && dist.getDistance() < 1000) {
            if ((actions.size() != 0 &&
                 !actions.get(actions.size() - 1).equals("500mMovingAway"))) {

                rule.fiveHundredMetersMovingAwayRule(dist.getDate());
                actions.add("500mMovingAway");
            }
            else if (!actions.get(actions.size() - 1).equals("200mMovingAway")
                     || actions.size() == 0) {

                rule.twoHundredMetersMovingAwayRule();
                actions.add("200mMovingAway");
                rule.fiveHundredMetersMovingAwayRule(dist.getDate());
                actions.add("500mMovingAway");
            }
        }
        // If the is 1000 or more meters home, activate
        // "oneThousandMetersApproachingRule"
        else if (dist.getDistance() >= 1000)
            if ((actions.size() != 0 &&
                 !actions.get(actions.size() - 1).equals("1000mMovingAway"))) {

                rule.oneThousandMetersMovingAwayRule(dist.getDate());
                actions.add("1000mMovingAway");
            }
            else if (!actions.get(actions.size()-1).equals("500mMovingAway")){

                rule.fiveHundredMetersMovingAwayRule(dist.getDate());
                actions.add("500mMovingAway");
                rule.oneThousandMetersMovingAwayRule(dist.getDate());
                actions.add("1000mMovingAway");
            }
            else if (!actions.get(actions.size() - 1).equals("200mMovingAway")
                     || actions.size() == 0){

                rule.twoHundredMetersMovingAwayRule();
                actions.add("200mMovingAway");
                rule.fiveHundredMetersMovingAwayRule(dist.getDate());
                actions.add("500mMovingAway");
                rule.oneThousandMetersMovingAwayRule(dist.getDate());
                actions.add("1000mMovingAway");
            }
        }

        // If user is at the same location more time,
        // apply all rules moving away from home
        else if (countEqualNumbers(aux, 0) == 4) {
            rule.twoHundredMetersMovingAwayRule();
            actions.add("200mMovingAway");
            rule.fiveHundredMetersMovingAwayRule(dist.getDate());
            actions.add("500mMovingAway");
            rule.oneThousandMetersMovingAwayRule(dist.getDate());
            actions.add("1000mMovingAway");
        }
    }
```

```
            // If user is more than 1 km home first time that initialize app,
            // apply all rules moving away from home
            else if (distances.size() != 0 && index == 0 &&
                     distances.get(index).getDistance() >= 1000) {

               rule.twoHundredMetersMovingAwayRule();
               actions.add("200mMovingAway");
               rule.fiveHundredMetersMovingAwayRule(dist.getDate());
               actions.add("500mMovingAway");
               rule.oneThousandMetersMovingAwayRule(dist.getDate());
               actions.add("1000mMovingAway");
            }

      }

      private static int countEqualNumbers(int[] array, int number) {
            int count = 0;
            for (int i = 0; i < array.length; i++)
                  if (array[i] == number)
                        count++;
            return count;
      }

}
```

**Figura 19 - Código fuente de la clase `UserBehaviour`**

#### 6.1.1.5. Rules

Las reglas que se aplican dependiendo del comportamiento del usuario están definidas en la clase Rules. Su implementación se muestra en siguiente código:

```
package upv.locamo.tfg.smarthome.logic;

import java.util.Calendar;
import java.util.Date;

import org.json.JSONException;
import org.json.JSONObject;
import org.restlet.ext.json.JsonRepresentation;
import org.restlet.representation.Representation;
import org.restlet.resource.ClientResource;

import upv.locamo.tfg.smarthome.server.ServerComponent;

public class Rules {

      /*
       * Some resources of smarthome
       */
      // Lightning - togglebistate functionality
      private String LIGHTS_KITCHEN_CENTRAL = "DF-CUINA.IL.CENTRAL";
      private String LIGHTS_KITCHEN_AUXILIARY = "DF-CUINA.IL.AUXILIAR";
      private String LIGHTS_KITCHEN_HOB = "DF-CUINA.IL.BANCADA";
      private String LIGHTS_DININGROOM_CENTRAL = "DF-MENJ.IL.CENTRAL";
      private String LIGHTS_DININGROOM_AUXILIARY = "DF-MENJ.IL.AUXILIAR";
```

```
	private String LIGHTS_ROOM1_CENTRAL = "DF-HAB1.IL.CENTRAL";
	private String LIGHTS_ROOM2_CENTRAL = "DF-HAB2.IL.CENTRAL";
	private String LIGHTS_ROOM3_CENTRAL = "DF-HAB3.IL.CENTRAL";

	// Blinds - movement functionality
	private String BLINDS_DININGROOM_12 = "DF-MENJ.FINESTRA.PERSIANES12";
	private String BLINDS_DININGROOM_34 = "DF-MENJ.FINESTRA.PERSIANES34";
	private String BLINDS_DININGROOM_5 = "DF-MENJ.FINESTRA.PERSIANA5";
	private String BLINDS_DININGROOM_ALL = "DF-MENJ.FINESTRA.PERSIANES";
	private String BLINDS_ROOM1 = "DF-HAB1.FINESTRA.PERSIANA";
	private String BLINDS_ROOM1_STOR = "DF-HAB1.FINESTRA.ESTOR";
	private String BLINDS_ROOM2 = "DF-HAB2.FINESTRA.PERSIANA";
	private String BLINDS_ROOM3 = "DF-HAB3.FINESTRA.PERSIANA";
	private String BLINDS_ROOM3_STOR = "DF-HAB3.FINESTRA.ESTOR";

	// Heating - togglebistate functionality
	private String HEATER_ROOM1 = "DF-HAB1.CL.CALEFACTOR";
	private String HEATER_ROOM2 = "DF-HAB2.CL.CALEFACTOR";
	private String HEATER_ROOM3 = "DF-HAB3.CL.CALEFACTOR";
	private String HEATER_CBATHROOM = "DF-BC.CL.CALEFACTOR";
	private String HEATER_DBATHROOM = "DF-BM.CL.CALEFACTOR";

	// Air conditioner - togglebistate functionality
	private String AIRCOND_DININGROOM = "DF-MENJ.CL.AC";
	private String AIRCOND_ROOM1 = "DF-HAB1.CL.AC";
	private String AIRCOND_ROOM3 = "DF-HAB3.CL.AC";

	// Sensors - bitstate functionality
	private String SENSOR_MOVEMENT = "DF-ENT.SE.DETMOV.SENSOR";
	private String SENSOR_CLOSEDOOR = "DF-PORTA.SENSOR.TANCADA";

	/*
	 * Actions than can be made over the above resources
	 */
	private String ENABLE = "biaON";
	private String DISABLE = "biaOFF";
	private String OPEN = "movaOPEN";
	private String CLOSE = "movaCLOSE";

	/**
	 * Rule that is activated when the user moves away 200 or more meters home
	 */
	public void twoHundredMetersMovingAwayRule() {

		System.out.println("Applying rule of 200 meters moving away");

		sendActionToServer(LIGHTS_KITCHEN_AUXILIARY, ENABLE);
		sendActionToServer(LIGHTS_DININGROOM_AUXILIARY, ENABLE);

		sendActionToServer(LIGHTS_KITCHEN_CENTRAL, DISABLE);
		sendActionToServer(LIGHTS_KITCHEN_HOB, DISABLE);
		sendActionToServer(LIGHTS_DININGROOM_CENTRAL, DISABLE);
		sendActionToServer(LIGHTS_ROOM1_CENTRAL, DISABLE);
		sendActionToServer(LIGHTS_ROOM2_CENTRAL, DISABLE);
		sendActionToServer(LIGHTS_ROOM3_CENTRAL, DISABLE);

		sendActionToServer(HEATER_CBATHROOM, DISABLE);
		sendActionToServer(HEATER_DBATHROOM, DISABLE);

		sendActionToServer(SENSOR_MOVEMENT, ENABLE);
		sendActionToServer(SENSOR_CLOSEDOOR, ENABLE);

	}
```

```
	/**
	 * Rule that is activated when the user approaches 200 or more meters home
	 */
	public void twoHundredMetersApproachingRule() {

		System.out.println("Applying rule of 200 meters approaching");

		sendActionToServer(LIGHTS_KITCHEN_AUXILIARY, ENABLE);
		sendActionToServer(LIGHTS_DININGROOM_AUXILIARY, ENABLE);

	}

	/**
	 * Rule that is activated when the user moves away 500 or more meters home
	 */
	public void fiveHundredMetersMovingAwayRule(Date date) {

		System.out.println("Applying rule of 500 meters moving away");

		// Down blinds only if we are in the months of May to October
		if (getMonth(date) >= 5 && getMonth(date) <= 10) {
			sendActionToServer(BLINDS_DININGROOM_12, CLOSE);
			sendActionToServer(BLINDS_DININGROOM_34, CLOSE);
			sendActionToServer(BLINDS_ROOM1, CLOSE);
			sendActionToServer(BLINDS_ROOM2, CLOSE);
			sendActionToServer(BLINDS_ROOM3, CLOSE);
			// Open that blinds if it is day
			if (getHour(date) <= 23 && getHour(date) >= 7) {
				sendActionToServer(BLINDS_DININGROOM_5, OPEN);
				sendActionToServer(BLINDS_ROOM1_STOR, OPEN);
				sendActionToServer(BLINDS_ROOM3_STOR, OPEN);
			}
			// Close that blinds if it is night
			else {
				sendActionToServer(BLINDS_DININGROOM_5, CLOSE);
				sendActionToServer(BLINDS_ROOM1_STOR, CLOSE);
				sendActionToServer(BLINDS_ROOM3_STOR, CLOSE);
			}
		}

	}

	/**
	 * Rule that is activated when the user approaches 500 or more meters home
	 */
	public void fiveHundredMetersApproachingRule(Date date) {

		System.out.println("Applying rule of 500 meters approaching");

		// Open blinds only if we are in the months of November to April
		if (getMonth(date) <= 4 && getMonth(date) >= 11) {
			// Open that blinds if it is day
			if (getHour(date) <= 23 && getHour(date) >= 7) {
				sendActionToServer(BLINDS_DININGROOM_ALL, OPEN);


				sendActionToServer(BLINDS_ROOM1_STOR, OPEN);
				sendActionToServer(BLINDS_ROOM2, OPEN);
				sendActionToServer(BLINDS_ROOM3_STOR, OPEN);
			}
		}

	}
```

```
/**
 * Rule that is activated when the user moves away 1000 or more meters home
 *
 * or when the user is a long time away from home
 */
public void oneThousandMetersMovingAwayRule(Date date) {

	System.out.println("Applying rule of 1000 meters moving away");

	// Turn off air conditioner only if we are in the months of June to
	// September
	if (getMonth(date) >= 6 && getMonth(date) <= 9) {
		sendActionToServer(AIRCOND_DININGROOM, DISABLE);
		sendActionToServer(AIRCOND_ROOM1, DISABLE);
		sendActionToServer(AIRCOND_ROOM3, DISABLE);
	}
	// Turn off heater only if we are in the months of November to
	// February
	else if (getMonth(date) >= 11 && getMonth(date) <= 2) {
		sendActionToServer(AIRCOND_DININGROOM, DISABLE);
		sendActionToServer(HEATER_ROOM1, DISABLE);
		sendActionToServer(HEATER_ROOM2, DISABLE);
		sendActionToServer(HEATER_ROOM3, DISABLE);
	}

}

/**
 * Rule that is activated when the user approaches 1000 or more meters home
 */
public void oneThousandMetersApproachingRule(Date date) {

	System.out.println("Applying rule of 1000 meters approaching");

	// Turn on air conditioner only if we are in the months of June to
	// September
	if (getMonth(date) >= 6 && getMonth(date) <= 9) {
		sendActionToServer(AIRCOND_DININGROOM, ENABLE);
		sendActionToServer(AIRCOND_ROOM1, ENABLE);
		sendActionToServer(AIRCOND_ROOM3, ENABLE);
	}
	// Turn on heater only if we are in the months of November to February
	else if (getMonth(date) >= 11 && getMonth(date) <= 2) {
		sendActionToServer(AIRCOND_DININGROOM, ENABLE);
		sendActionToServer(HEATER_ROOM1, ENABLE);
		sendActionToServer(HEATER_ROOM2, ENABLE);
		sendActionToServer(HEATER_ROOM3, ENABLE);
	}

}

/**
 * Sends to server an action on a given resource
 * @param resource
 * @param action
 */
private void sendActionToServer(String resource, String action) {

	ClientResource clientResource = new ClientResource(
		ServerComponent.getSmartHomeURL() + "/devFunc/" + resource);
```

```
		try {
			JSONObject jsonSend = new JSONObject();
			jsonSend.put("action", action);
			Representation obj = new JsonRepresentation(jsonSend);
			clientResource.put(obj);
		} catch (JSONException e) {
			e.printStackTrace();
		}
	}


	/**
	 * Returns the month of a given date
	 * @param date
	 * @return month (int)
	 */
	private int getMonth(Date date) {
		Calendar calendar = Calendar.getInstance();
		calendar.setTime(date);
		return calendar.get(Calendar.MONTH);
	}

	/**
	 * Returns the hour of a given date
	 * @param date
	 * @return month (int)
	 */
	private int getHour(Date date) {
		Calendar calendar = Calendar.getInstance();
		calendar.setTime(date);
		return calendar.get(Calendar.HOUR_OF_DAY);
	}

}
```

**Figura 20 - Código fuente de la clase `Rules`**

### 6.1.2. Creación del servidor

Para la creación de un servidor en Restlet son necesarios dos componentes: `Component` y `Application`. `Component` será el contenedor de `Application` y éste último será el encargado de la gestión de las peticiones HTTP.

#### 6.1.2.1. `ServerComponent`

La clase `ServerComponent` contendrá los parámetros de configuración del servidor, indicando el protocolo y el puerto. En este caso interesa que sea un servidor HTTP, que atenderá las peticiones en el puerto 8284; se ha decidido este número de puerto porque el servidor SmartHome utiliza el puerto 8182.

Cuando se ejecute esta clase, se pedirá al usuario que introduzca la dirección y el puerto del servidor SmartHome debido a que existe la posibilidad de que no estén situados físicamente en la misma máquina.

Una vez creado el servidor y definida la URL del servidor SmartHome, corresponde indicar que este `Component` es el contenedor de la `Application`. Además se ha de añadir un cliente para poder realizar peticiones al servidor de la vivienda inteligente.

```
package upv.locamo.tfg.smarthome.server;

import java.util.Scanner;
import org.restlet.Client;
import org.restlet.Component;
import org.restlet.Context;
import org.restlet.data.Protocol;

public class ServerComponent extends Component{

	private static String url = "";
	private static String port = "";

	public static String getSmartHomeURL(){
		return "http://" + url + ":" + port;
	}

	public static void main (String[] args) throws Exception{
		new ServerComponent().start();
	}

	private void defineSmartHomeServer(){
	 System.out.println ("Please introduce the URL and port of SmartHome Server
	                      (separated by a space)");
	 String inputData = "";
	 Scanner scanner = new Scanner (System.in);
	 inputData = scanner.nextLine();
	 if (!inputData.contains(" ")){
		System.out.println ("Please introduce it again (incorrect format)");
		inputData = scanner.nextLine();
	 }
	 String[] parts = inputData.split(" ");
		url = parts[0];
		port = parts[1];
		scanner.close();
	}

	public ServerComponent(){
		// Define the URL for the SmartHome Server
		defineSmartHomeServer();
		// Add a new Server at port 8284
		getServers().add(Protocol.HTTP, 8284);
		// Attach the application to the default virtual host
		getDefaultHost().attachDefault(new ServerApplication());
		// Add a client to do calls to SmartHome server
		getClients().add(new Client(new Context(), Protocol.HTTP));
	}
}
```

**Figura 21 - Código fuente de la clase `ServerComponent`**

#### 6.1.2.2. ServerApplication

Esta clase será la encargada de dirigir las peticiones HTTP que reciba el servidor al correspondiente recurso según la URL; para ello, se sobrescribe el método `createInboundRoot()` utilizando la clase `Router` que proporciona Restlet. Los atributos entre corchetes indican una variable que puede tener cualquier valor; por ejemplo si el usuario introduce http://localhost:8284/users/locamo, el usuario (`userID`) será locamo.

```
package upv.locamo.tfg.smarthome.server;

import org.restlet.Application;
import org.restlet.Restlet;
import org.restlet.routing.Router;

import upv.locamo.tfg.smarthome.server.resources.DistancesServerResource;
import upv.locamo.tfg.smarthome.server.resources.ShoppingItemServerResource;
import upv.locamo.tfg.smarthome.server.resources.ShoppingListServerResource;
import upv.locamo.tfg.smarthome.server.resources.UserServerResource;
import upv.locamo.tfg.smarthome.server.resources.UsersServerResource;

public class ServerApplication extends Application{

	/**
	 * Constructor for change the settings
	 */
	public ServerApplication() {
		setName("TFG Location Server");
		setDescription("Server implemented for the development of TFG");
		setOwner("Lorena Calabuig");
		setAuthor("locamo");
	}

	/**
	 * Factory method called by the framework when the application starts
	 * @return Restlet
	 */
	@Override
	public Restlet createInboundRoot() {

		Router router = new Router(getContext());
		router.attach("/", RootServerResource.class);
		router.attach("/users", UsersServerResource.class);
		router.attach("/users/{userID}", UserServerResource.class);
		router.attach("/distances", DistancesServerResource.class);
		router.attach("/shoppingList", ShoppingListServerResource.class);
		router.attach("/shoppingList/{item}",
ShoppingItemServerResource.class);

		return router;
	}

}
```

Figura 22 - Código fuente de la clase `ServerApplication`

### 6.1.3. Recursos del servidor

Dependiendo de la URL de las peticiones HTTP, el `Router` las dirigirá a un recurso del servidor u otro. Estos recursos utilizan las clases cuya implementación se ha mostrado en el subapartado 7.1.1.

Para atender las peticiones HTTP que recibe el servidor, es necesario sobrescribir las subrutinas que implementa Restlet para tal fin, de forma que, si se quiere responder cierta información en un determinado formato cuando se envía una petición GET, se tendrá que sobrescribir el método `get()` y devolver la información en forma de `Representation`.

En el siguiente punto se va a mostrar la implementación de una clase completa para un recurso del servidor (`ServerResource`) por lo que, tomando ésta como base, en los siguientes puntos referentes a los recursos, sólo se mostrarán los métodos más relevantes para su desarrollo.

#### 6.1.3.1. `UsersServerResource`

Este recurso será el encargado de gestionar una lista con todos los usuarios registrados en el servidor; gestionará los métodos GET y PUT en los que se podrá, respectivamente, generar un `JSONArray` con todos los usuarios junto con lista con cada una de sus posiciones, y añadir un nuevo usuario, que como resultado devolverá el usuario añadido en formato JSON.

En esta clase se implementan también algunos métodos auxiliares para manejar la mencionada lista.

```
package upv.locamo.tfg.smarthome.server.resources;

import java.io.IOException;
import java.util.ArrayList;

import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
import org.restlet.data.Status;
import org.restlet.ext.json.JsonRepresentation;
import org.restlet.representation.Representation;
import org.restlet.resource.ResourceException;
import org.restlet.resource.ServerResource;

import upv.locamo.tfg.smarthome.logic.Location;
import upv.locamo.tfg.smarthome.logic.User;

public class UsersServerResource extends ServerResource {


	private static ArrayList<User> users = new ArrayList<User>();
```

```
	private String password = "tfg";

	public static ArrayList<User> getUsers() {
		return users;
	}

	/**
	 * Method GET: gets the list with all users registered and their locations
	 */
	@Override
	protected Representation get() throws ResourceException {

		JSONArray jsonArray = new JSONArray();
		try {
			for (int i = 0; i < getUsers().size(); i++) {
				jsonArray.put(getUsers().get(i).getJSONUser());
			}
			return new JsonRepresentation(jsonArray);
		} catch (JSONException e) {
			setStatus(Status.CLIENT_ERROR_NOT_FOUND);
			return null;
		}
	}

	/**
	 * Method PUT: inserts a new user
	 */
	@Override
	protected Representation put(Representation representation) throws
ResourceException {

		try {
			JSONObject json =
			     (new JsonRepresentation(representation)).getJsonObject();
			String username = json.getString("userID");
			String pass = json.getString("pass");
			// Adds a user only if he/she introduced the correct password
			if (pass.equals(password)){
				User user = new User(username,
							new ArrayList<Location>());
				getUsers().add(user);
				return new JsonRepresentation(json);
			}
			else {
				setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
				return null;
			}
		} catch (JSONException | IOException e) {
			setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
			return null;
		}

	}
```

```
	/**
	 * Obtains the list with the locations of a specific user
	 * @param uname
	 * @return list with locations
	 */
	public static ArrayList<Location> getLocationListByUser (String uname){
		ArrayList<Location> list = new ArrayList<Location>();
		for (User user : users){
			if (user.getUsername().equals(uname))
				list = user.getLocationList();
		}
		return list;
	}

	/**
	 * Check if an user is already inserted in the list
	 * @param uname
	 * @return
	 */
	public static boolean contains (String uname){
		for (User user : users){
			if (user.getUsername().equals(uname))
				return true;
		}
		return false;
	}

	/**
	 * Adds a Location in the list only if the location received is not
	 * the same as an already inserted
	 * @param uname
	 * @param location
	 * @return false if location not exists in the list
	 */
	public static boolean ifUserHaveLocation(String uname, Location l){
		ArrayList<Location> list = getLocationListByUser(uname);
		for (Location loc: list){
			if (loc.getTime() == l.getTime()){
				return true;
			}
		}
		return false;
	}

}
```

**Figura 23 - Código fuente del recurso de a lista de usuarios**

Si un cliente enviara una petición GET al servidor para obtener la lista de los usuarios registrados en el servidor, obtendría un array JSON como el que se muestra en la figura 24. En cambio, si el servidor recibiese una petición PUT para añadir un usuario, la respuesta que obtendría el cliente, además de añadirlo a la lista de usuarios, sería la que se muestra en la figura 25.

```
[
    {
       "userID": "locamo",
        "location":
        [
           {
             "date": "Tue Aug 12 21:23:39 CEST 2014",
             "latitude": 38.7181938,
             "accuracy": 21,
             "longitude": -0.6662247
           },
           {
             "date": "Tue Aug 12 21:41:10 CEST 2014",
             "latitude": 38.71713323,
             "accuracy": 53,
             "longitude": -0.65893525
           }
        ]
    },
    {
       "userID": "roales",
        "location":
        [
           {
             "date": "Tue Aug 12 20:48:03 CEST 2014",
             "latitude": 39.4747223,
             "accuracy": 21,
             "longitude": -0.3416705
           }
        ]
    }
 ]
```


**Figura 24 – Ejemplo de representación de la lista de usuarios en formato JSON**


```
    {
        "userID": "nuevo_usuario",
        "locaton":
          [
          ]

     }
```


**Figura 25 – Ejemplo de representación del nuevo usuario insertado en JSON**

### 6.1.3.2. UserServerResource

Si un cliente añadir una nueva localización a un usuario o simplemente obtener la lista con sus localizaciones, el recurso `UserServerResource`, será el encargado de recibir dichas peticiones. Para ello, es necesario obtener el nombre de usuario de la URL introducida de la siguiente forma:

```
	private String userID;

	/**
	 * This method gets the user by the URL Example: URL:
	 *      http://localhost:8284/users/locamo
	 *      user = locamo
	 */
	@Override
	protected void doInit() throws ResourceException {
		userID = getAttribute("userID");


	}
```

**Figura 26 - Método de la clase `UserServerResource` para obtener el usuario**

El método que implementa la respuesta a la petición GET se ilustra a continuación y devolverá un resultado como el mostrado en la figura 15.

```
	@Override
	protected Representation get() throws ResourceException {
		if (UsersServerResource.contains(userID)) {
			ArrayList<Location> arrayList = UsersServerResource
					.getLocationListByUser(userID);
			User user = new User(userID, arrayList);
			try {
				return new JsonRepresentation(user.getJSONUser());
			} catch (JSONException e) {
				e.printStackTrace();
			}
		} else {
			setStatus(Status.CLIENT_ERROR_NOT_FOUND,
				"Invalid username");
			return null;
		}

	}
```

**Figura 27 - Método GET de la clase `UserServerResource`**

Si el cliente desea introducir una nueva localización a la lista de las geoposiciones de un usuario, el método que recibe dicha petición es el ilustrado en la figura 28.

```
@Override
 protected Representation put(Representation rep) throws ResourceException {

        try {
                JSONObject json =
                   (new JsonRepresentation(rep)).getJsonObject();
                JSONArray jsonArray = new JSONArray();
                jsonArray = json.getJSONArray("location");
                User user = new User(userID,
                    UsersServerResource.getLocationListByUser(userID));

                // Add the location sent to the locations list and
                // calculate distance
                Distance dist = addLocationToList(user, jsonArray);
                UserBehaviour.applyRulesAccordingDistance(dist);

                return new JsonRepresentation(new User(userID,
                              user.getLocationList()).getJSONUser());

        } catch (JSONException | IOException e) {
                setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
                return null;
        }

}
```

**Figura 28 - Método PUT de la clase `UserServerResource`**

Como puede observarse, al mismo tiempo que se inserta una nueva posición, se calcula la distancia que hay desde ese punto hasta la vivienda y se aplica, si procede –dependiendo del algoritmo definido en la clase `UserBehaviour` –, una de las reglas definidas.

#### 6.1.3.3. `DistancesServerResource`

Esta clase sirve para obtener la lista con todas las distancias calculadas. A continuación se muestra el código del método `get()` para tal fin.

```java
@Override
protected Representation get() throws ResourceException {

	JSONArray jsonArray = new JSONArray();

	try {
		for (int i = 0; i < UserBehaviour.getDistancesList().size();
		     i++) {
		       JSONObject jsonObj = new JSONObject();
		       Distance dist = UserBehaviour.getDistancesList().get(i);
		       jsonObj.put("time", dist.getDate());
		       jsonObj.put("distance", dist.getDistance());
		       jsonObj.put("accuracy", dist.getAccuracy());
		       jsonArray.put(jsonObj);
		}
		return new JsonRepresentation(jsonArray);
	} catch (JSONException e) {
		e.printStackTrace();
		return new StringRepresentation("Cannot create the list");
	}

}
```

**Figura 29 - Método GET de la clase `DistancesServerResource`**

La respuesta que se enviará al cliente al invocar este método es un array JSON con las distancias desde una posición dada hasta la casa que se hayan calculado, la precisión que puede proporcionar el dispositivo proveedor de la posición y la fecha de la misma. A continuación se muestra un ejemplo:

```json
[
    {
       "distance": 56.7894256,
       "accuracy": 23,
       "date": " Tue Aug 12 21:23:39 CEST 2014"
    },
    {
       "distance": 47.23576878,
       "accuracy": 15,
       "date": " Tue Aug 12 21:27:37 CEST 2014"
    }
]
```

**Figura 30 - Ejemplo de representación de la lista con las distancias en formato JSON**

#### 6.1.3.4. ShoppingListServerResource

Este recurso del servidor va a implementar la lista de la compra. El cliente podrá solicitar ver todos los elementos existentes en la lista, mediante el método GET y cuyo código se muestra en la Figura 31; y añadir nuevos ítems, mediante el método PUT, cuya implementación se plasma en la Figura 32.

```
@Override
protected Representation get() throws ResourceException {

        return new JsonRepresentation(shoppingListToJSON());

}
```

**Figura 31 - Método GET de la clase ShoppingListServerResource**

El método `shoppingListToJSON()` que se utiliza en el método GET de la figura anterior, crea un objeto JSON cuyo valor es un array con todos los elementos que hay en la lista de la compra. A continuación se muestra cómo quedaría la representación de la lista de la compra:

```
{
     "shopping_list":
     [
        "pollo",
        "agua",
        "aceite"
     ]
 }
```

**Figura 32 - Ejemplo de representación de la lista de la compra en formato JSON**

En caso de querer introducir un nuevo elemento, se deberá invocar el método PUT enviando un objeto JSON que contendrá un array con los ítems a insertar. La respuesta que obtendrá el cliente seguirá el mismo formato que el mostrado en la Figura 32. A continuación se muestra el código de la implementación de este método:

```
@Override
protected Representation put(Representation representation)
		throws ResourceException {

	try {
		JSONObject json = (new JsonRepresentation(representation))
				.getJsonObject();
		JSONArray jsonArray = new JSONArray();
		jsonArray = json.getJSONArray("shopping_list");
		for (int i = 0; i < jsonArray.length(); i++) {
			if (!ifItemExists(jsonArray.get(i).toString()))
				shoppingList.add(jsonArray.get(i).toString());
		}

		return new JsonRepresentation(shoppingListToJSON());

	} catch (JSONException | IOException e) {
		setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
		return null;
	}


}
```

**Figura 33 - Método PUT de la clase ShoppingListServerResource**

### 6.1.3.5. ShoppingItemServerResource

El último recurso del servidor implementado es el ShoppingItemServerResource, que únicamente se utiliza para eliminar un elemento concreto de la lista de la compra cuando el usuario ya lo ha adquirido. Para tal fin, se ha de implementar el método delete() y se va a utilizar el mismo método que el que se muestra en la Figura 26 para obtener el "item" a eliminar de la URL.

```
@Override
protected Representation delete() throws ResourceException {
	ArrayList<String> aux = ShoppingListServerResource.getShoppingList();
	if (aux.size() != 0){
		aux.remove(item);
		return new
	   JsonRepresentation(ShoppingListServerResource.shoppingListToJSON());
	}
	else {
		setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
		return null;
	}

}
```

**Figura 34 - Método DELETE de la clase ShoppingItemServerResource**

## 6.2. Aplicación móvil basada en Android

Una vez expuesta la implementación del servidor, se expondrá a continuación la de la aplicación móvil basada en Android. Se ha procurado ceñirse a las especificaciones descritas en el subapartado 5.2.2. y, además, se ha traducido al inglés.

En este apartado se desarrollará tanto la parte lógica – escrita en Java – como la parte gráfica – escrita en XML – de cada `Activity` de la aplicación. Se entiende como `Activity` cada una de las pantallas sobre las que se moverá el usuario para realizar las diferentes acciones definidas en la etapa de análisis.

No es objetivo de este capítulo volver a describir cuál es la funcionalidad de cada una de estas *activities*, así que se ilustrará solamente el código Java y la interfaz gráfica asociada a cada una de ellas y se explicará cuál es la funcionalidad de ciertos métodos.

### 6.2.1. MenuActivity

Esta actividad será la que aparecerá siempre cada vez que el usuario abra la aplicación y desde la que se accederá a la pantalla principal con el mapa.

A continuación se muestra la implementación completa de esta actividad como ejemplo base sobre el que se construirán el resto de clases y, de esta forma, no será necesario mostrarlo en los siguientes *activities*. método `onCreate()` de esta actividad, que es método que se ejecuta cada vez que se llama a una actividad.

```
package upv.locamo.tfg.smarthome.app;

import android.content.Intent;
import android.content.SharedPreferences;
import android.preference.PreferenceManager;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.Button;

import upv.locamo.tfg.smarthome.app.utils.Utils;


public class MenuActivity extends ActionBarActivity {

    SharedPreferences appPreferences;
    boolean isAppInstalled = false;

    private Button btn_start;
    private Button btn_exit;
```

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

        btn_start = (Button) findViewById(R.id.btn_start);
        btn_exit = (Button) findViewById(R.id.btn_exit);

        // Start application
        btn_start.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                // If the user is not logged in to the application
                if (Utils.getUser(getApplicationContext()).equals("user")) {
                    Intent loginActivity = new Intent(MenuActivity.this,
LoginActivity.class);
                    startActivity(loginActivity);
                }
                else {
                    Intent mainActivity = new Intent(MenuActivity.this,
MainActivity.class);
                    startActivity(mainActivity);
                }
            }
        });

        // Exit
        btn_exit.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                finish();
                System.exit(0);
            }
        });
    }


    @Override
    public boolean onCreateOptionsMenu(Menu menu) {

        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu, menu);
        return true;
    }

    /**
     * Adds a shortcut for the application on home screen
     */
    private void addShortcut() {

        appPreferences = PreferenceManager.getDefaultSharedPreferences(this);
        isAppInstalled = appPreferences.getBoolean("isAppInstalled",false);
        if(isAppInstalled==false) {
            Intent shortcutIntent = new Intent(getApplicationContext(),
                    MenuActivity.class);

            shortcutIntent.setAction(Intent.ACTION_MAIN);

            Intent addIntent = new Intent();
            addIntent
                    .putExtra(Intent.EXTRA_SHORTCUT_INTENT, shortcutIntent);
            addIntent.putExtra(Intent.EXTRA_SHORTCUT_NAME, "My SmartHome");
```

```
            addIntent.putExtra(Intent.EXTRA_SHORTCUT_ICON_RESOURCE,
                    Intent.ShortcutIconResource.fromContext(getApplicationContext(),
                            R.drawable.ic_launcher)
            );

            addIntent
                    .setAction("com.android.launcher.action.INSTALL_SHORTCUT");
            getApplicationContext().sendBroadcast(addIntent);

            SharedPreferences.Editor editor = appPreferences.edit();
            editor.putBoolean("isAppInstalled", true);
            editor.commit();
        }
    }


    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();
        if (id == R.id.about) {
            Intent aboutActivity = new Intent(MenuActivity.this,
AboutUsActivity.class);
            startActivity(aboutActivity);
        }
        return super.onOptionsItemSelected(item);
    }

}
```

**Figura 35 – Implementación de la clase `MenuActivity`**

El método `addShortcut()` se utiliza para que cuando se instale la aplicación, se cree un acceso directo en la pantalla principal del dispositivo. Se hace uso de las preferencias (`SharedPreferences`), que es un archivo en el que se guardan variables que se desean conservar para personalizar la aplicación; un ejemplo de ello sería mantener información acerca del usuario (nombre, contraseña, etc.) o, como este es el caso, escribir en dicho archivo que la aplicación está ya instalada y por tanto no instalará otro acceso directo.

Por lo que respecta a la parte del desarrollo de la interfaz, se ha intentado que sea lo más parecido posible al *mockup* diseñado.


**Figura 36 - Interfaz gráfica del menú principal**

### 6.2.2. LoginActivity

En esta actividad será donde el usuario podrá registrarse en la aplicación para poder acceder a ella. Para registrarse, hay que enviar comprobar primero en el servidor TFGLocation si existe ya el usuario; en caso de que no existiera, hay que insertarlo en la lista de usuarios. Además, hay que guardar el usuario a nivel local de la aplicación, por lo que se utilizan las `SharedPreferences`, cuyo fichero de preferencias se modifica en la clase `Utils`. En el método `onCreate()` se hace todo este trabajo.

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

    btn_register = (Button) findViewById(R.id.btn_register);
    et_user = (EditText) findViewById(R.id.et_user);
    et_pass = (EditText) findViewById(R.id.et_password);

    btn_register.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            user = et_user.getText().toString();
            pass = et_pass.getText().toString();
            if (!user.matches("") && !pass.matches("")) {
```

```
                    sendUserToServer();
                    if (ifUserIsAdded()) {
                        updateSharedPreferences();
                        goToMainActivity();
                    } else {
                        Toast.makeText(getApplicationContext(), "Please enter a
valid username/password", Toast.LENGTH_SHORT).show();
                    }
                } else {
                    Toast.makeText(getApplicationContext(), "Please enter a valid
username/password", Toast.LENGTH_SHORT).show();
                }
            }
        });
    }
```

**Figura 37 - Código del método `onCreate()` de la clase `LoginActivity`**

El método `onCreate()` hace uso de otros métodos que son: `sendUserToServer()`, `ifUserIsAdded()`, `updateSharedPreferences()` y `goToMainActivity()`. En los siguientes subapartados se mostrará una breve descripción de su funcionalidad y la implementación de cada uno de ellos.

#### 6.2.2.1. `sendUserToServer()`

Este método sirve para ejecutar una `AsyncTask` (tarea asíncrona o en segundo plano, que se ejecuta en otro hilo diferente al principal) que se encargará de comprobar si el nombre de usuario (`user`) introducido existía ya en el servidor TFGLocation (`url`) o no. Si el usuario existe, el servidor devolverá el código de estado 200; si no existe, devolverá el código 404.

```
    public void sendUserToServer() {
        if (!userExists()) {
            SendUserToServerTask sendUsertoServerTask = new SendUserToServerTask();
            sendUsertoServerTask.execute();
        }
    }

    /**
     * AsyncTask for send the new user to server
     */
    private class SendUserToServerTask extends AsyncTask<Void, Void, Void> {
        @Override
        protected Void doInBackground(Void... params) {
            ClientResource resource = new ClientResource(url);

            try {
                JSONObject jsonSend = new JSONObject();
                jsonSend.put("userID", user);
                jsonSend.put("ip", "null");
                jsonSend.put("pass", pass);
```

```
                Representation obj = new JsonRepresentation(jsonSend);
                resource.put(obj);
            } catch (JSONException e) {
                e.printStackTrace();
            }
            return null;
        }
    }

    private boolean userExists() {
        boolean retval = false;
        try {
            CheckIfUserExistsTask checkIfUserExistsTask = new
    CheckIfUserExistsTask();
            retval = checkIfUserExistsTask.execute(url + user).get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        return retval;
    }

    /**
     * AsyncTask for send to server the new user
     */
    private class CheckIfUserExistsTask extends AsyncTask<String, Void, Boolean> {
        @Override
        protected Boolean doInBackground(String... params) {
            try {
                URL url = new URL(params[0]);
                URLConnection connection = url.openConnection();
                connection.connect();
                HttpURLConnection httpConnection = (HttpURLConnection) connection;
                int code = httpConnection.getResponseCode();
                if (code != 200) {
                    return false;
                } else {
                    return true;
                }
            } catch (IOException e) {
                return false;
            }

        }

    }
```

**Figura 38 - Métodos para registrar un nuevo usuario en el servidor TFGLocation**

#### 6.2.2.2. ifUserIsAdded()

Este método comprueba que se ha añadido correctamente el usuario en el servidor. Para ello, hace una llamada al método `userExists()` de la Figura 38; si no se ha introducido correctamente la contraseña, no se ha podido registrar en el servidor y,

por tanto, aparecerá un mensaje en la pantalla indicando que el usuario o la contraseña no son correctos.

```
private boolean ifUserIsAdded() {
    return userExists();
}
```

Figura 39 - Método `ifUserIsAdded()` de la clase `LoginActivity`

#### 6.2.2.3. updateSharedPreferences()

Este método se utiliza para guardar el nombre del usuario introducido en la aplicación y hace uso de la clase `Utils`, que se adjunta en el Anexo B, apartado 1.

```
private void updateSharedPreferences() {
    Utils.setUser(getApplicationContext(), user);
}
```

Figura 40 - Método `updateSharedPreferences()` de la clase `LoginActivity`

#### 6.2.2.4. goToMainActivity()

Por último, y una vez concluido el registro, se pasará a la pantalla principal de la aplicación en la que aparecerá el mapa con distintos marcadores.

```
private void goToMainActivity() {
    Intent mainActivity = new Intent(LoginActivity.this, MainActivity.class);
    startActivity(mainActivity);
}
```

Figura 41 - Método `goToMainActivity()` de la clase `LoginActivity`

La interfaz para el registro del usuario desarrollada se ilustra en la siguiente figura.


**Figura 42 - Interfaz gráfica de la pantalla de registro**

### 6.2.3. MainActivity

La clase `MainActivity` es en la que se mostrará el mapa con el marcador de la posición en la que se encuentra la vivienda, la posición actual del usuario y los supermercados más cercanos al usuario en caso de tener ítems en la lista de la compra.

La clase `DevicePosition` es una clase auxiliar que, a grandes rasgos, tiene la siguiente funcionalidad: obtiene la posición actual del usuario y la muestra en el mapa; crea y envía al servidor un objeto en formato JSON con la posición obtenida; y obtiene los supermercados más cerca del usuario en base a esa posición, los muestra en el mapa y crea una notificación. En el Anexo B, apartado 2, se describe toda la funcionalidad de esta clase y se muestra su implementación.

```
public static GoogleMap map;
private LatLng HOME;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    context = getApplicationContext();
    System.setProperty("java.net.preferIPv6Addresses","false");

    user = Utils.getUser(context);
```

```
        getCurrentLocation();

        HOME = new LatLng(38.717338,-0.6591331);

        // Get the map
        map = ((SupportMapFragment)
    getSupportFragmentManager().findFragmentById(R.id.map)).getMap();

        // Move camera to home location
        CameraPosition cameraPositionHome = new CameraPosition.Builder()
                .target(HOME)   // Center map at home
                .zoom(17)       // Zoom
                .build();
        CameraUpdate cameraUpdateHome =
                CameraUpdateFactory.newCameraPosition(cameraPositionHome);
        map.animateCamera(cameraUpdateHome);

        // Add a marker to map that indicates home location
        map.addMarker(new MarkerOptions()
                .position(HOME)
                .title(getApplicationContext().getString(R.string.home)));

    }
```

**Figura 43 - Método `onCreate()` de la clase `MainActivity`**

En el método `onCreate()` de la figura anterior, se obtiene el mapa que se ha insertado en el XML para poder añadir marcadores de posición y centrar el mapa en una localización en concreto; en este caso, se quiere que la vista del mapa se centre en la dirección de la casa y añada un marcador indicándolo.

Como se ha descrito anteriormente, en el mapa también se muestra la posición del usuario cada vez que se actualiza y, en caso de haber elementos en la lista de la compra, la aplicación buscará y mostrará en el mapa los lugares más cercanos alrededor de la posición del usuario en los que pueda realizar la compra.

El menú de esta actividad tendrá las siguientes opciones: “Lista de la compra”, “Acerca de” y “Cerrar sesión”.

El método que implementa este menú se muestra a continuación:

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {

        int id = item.getItemId();
        if (id == R.id.about) {
            Intent aboutActivity = new Intent(MainActivity.this,
                                               AboutUsActivity.class);
            startActivity(aboutActivity);
        }
```

```
        if (id == R.id.logout) {
            Intent logoutActivity = new Intent(this, LogoutActivity.class);
            logoutActivity.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP |
                    Intent.FLAG_ACTIVITY_CLEAR_TASK |
                    Intent.FLAG_ACTIVITY_NEW_TASK);
            startActivity(logoutActivity);
            finish();
        }
        if (id == R.id.shoppingList) {
            Intent shoppingListActivity = new Intent(MainActivity.this,
    ShoppingListActivity.class);
            startActivity(shoppingListActivity);
        }
        return super.onOptionsItemSelected(item);

    }
```

Figura 44 - Método para mostrar opciones en el menú

Las siguientes ilustraciones muestran las distintas situaciones que se pueden dar:

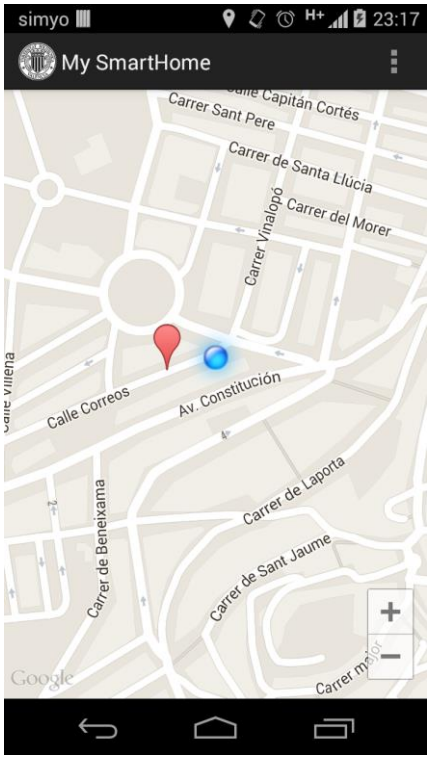



Figura 45 - Pantalla principal con la posición de la casa y la del usuario

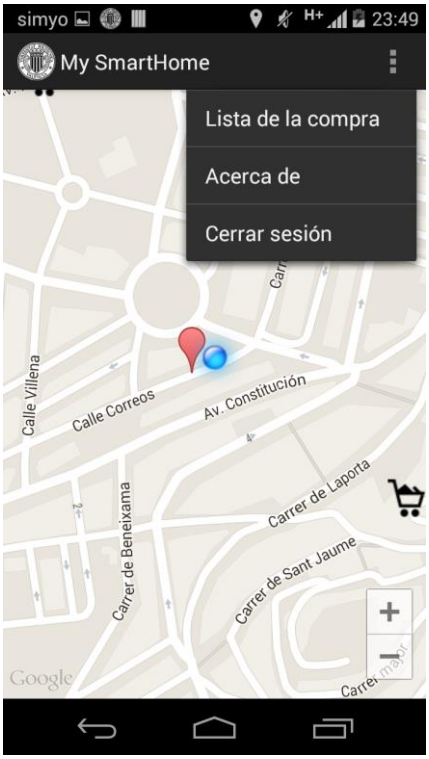



Figura 46 - Pantalla principal con las opciones del menú

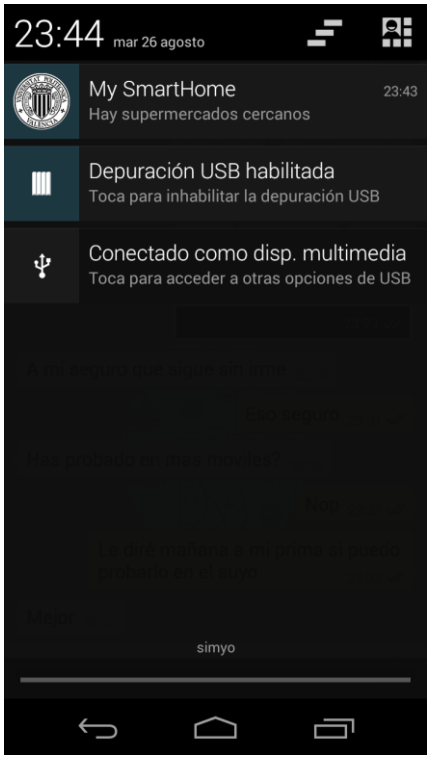



Figura 47 - Notificación que recibe el usuario cuando hay supermercados cerca

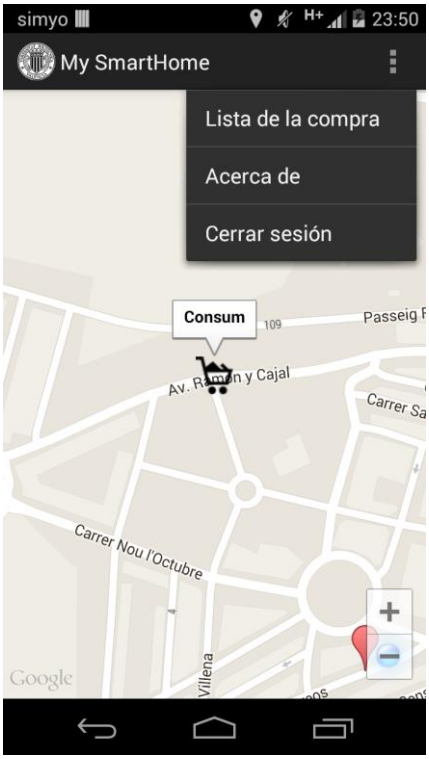



Figura 48 - Pantalla principal con los supermercados cerca y su nombre

### 6.2.4. ShoppingListActivity

La actividad ShoppingListActivity es la encargada de gestionar la introducción y eliminación de elementos de la lista de la compra y de su sincronización con el servidor.

La gestión de una lista con ítems, requiere de la creación de dos layouts: activity_shopping_list, en el que se mostrará un campo para añadir el nombre del elemento y el correspondiente botón de "Añadir", y la lista con mencionados elementos; y list_row, que tendrá los elementos de cada uno de los ítems de la lista. Además se tendrá que crear un Adapter para mostrar los elementos de la lista en la pantalla.

La pantalla mostrará una lista con todos los productos que haya que comprar si la lista de la compra no está vacía en el servidor; en caso contrario, no mostrará nada. El usuario podrá añadir nuevos elementos introduciendo el nombre del producto y pulsando el botón "Añadir". Para eliminar un producto de la lista, tendrá que pulsar durante un segundo sobre mencionado producto. Cada una de las acciones realizadas se enviarán como petición al servidor (métodos GET, PUT y DELETE, respectivamente).

```
private ListView lv_shopping;
public ArrayAdapter<String> adapter;
private SmarthomeShoppingList smarthomeShoppingList;

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

    System.setProperty("java.net.preferIPv6Addresses", "false");

    […]

    smarthomeShoppingList = new SmarthomeShoppingList(this);
    smarthomeShoppingList.getShoppingListFromServer();

    lv_shopping.setAdapter(adapter);
    btn_addItem.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            String itemToAdd = et_item.getText().toString();
            if (!itemToAdd.matches("")) {
                String aux = itemToAdd.substring(0, 1).toUpperCase() +
itemToAdd.substring(1);
                adapter.add(aux);
                smarthomeShoppingList.addItemToShoppingList(aux);
                et_item.setText("");
            }
        }
    });

    lv_shopping.setOnItemLongClickListener(new
AdapterView.OnItemLongClickListener() {
        @Override
        public boolean onItemLongClick(AdapterView<?> arg0, View arg1,
                                       int position, long arg3) {

            String itemToDelete =
                 lv_shopping.getItemAtPosition(position).toString();
            smarthomeShoppingList.deleteItemFromShoppingList(itemToDelete);
            adapter.remove(lv_shopping.getItemAtPosition(position).toString());
            adapter.notifyDataSetChanged();
            adapter.notifyDataSetInvalidated();

            return true;
        }
    });

}
```

**Figura 49 - Método `onCreate()` de la clase `ShoppingListActivity`**

Esta actividad hace llamadas a la clase `SmarthomeShoppingList`, que es la encargada de obtener la lista con los elementos a comprar y añadir y eliminar elementos del servidor; se describe en el Anexo B, apartado 3.

La siguiente captura de pantalla muestra la interfaz final para gestionar la lista de la compra.

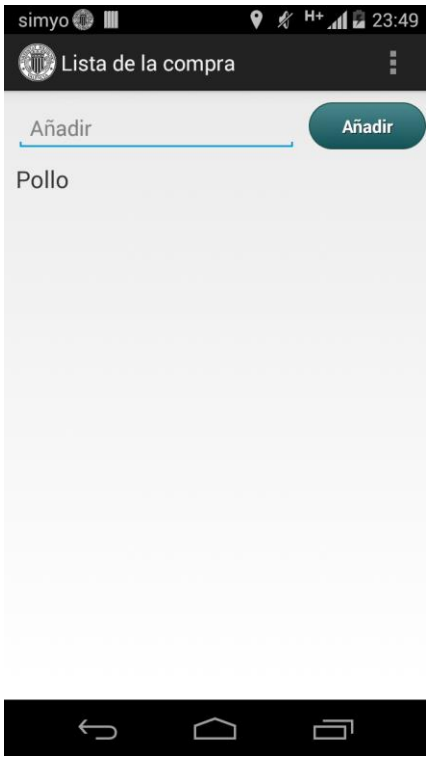



**Figura 50 - Interfaz de usuario para la lista de la compra**

### 6.2.5. AboutUsActivity

Esta pantalla sólo muestra la información acerca de la aplicación: propósito, autor y tutor del trabajo. A continuación se muestra una captura de la pantalla:

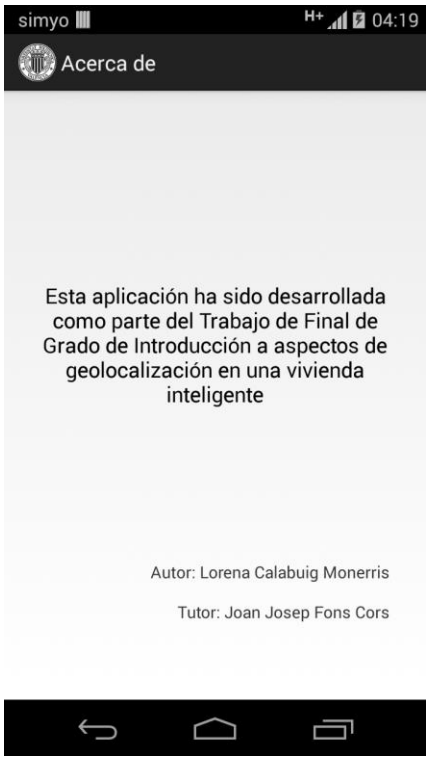



Figura 51 - Interfaz de "Acerca de"

### 6.2.6. LogoutActivity

Si el usuario selecciona la opción de cerrar sesión en el menú de la pantalla principal, todas las actividades que había en la pila de actividades se finalizan y se elimina el usuario actual del archivo de preferencias de la aplicación.

Cuando el usuario finaliza sesión, le lleva directamente a la pantalla inicial para o bien empezar sesión con otro usuario, o bien finalizar la aplicación.

El siguiente código implementa las acciones descritas anteriormente:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Delete the current user from the Shared Preferences
    Utils.deleteCurrentUser(MainActivity.getContext());

    Intent menuActivity = new Intent(this, MenuActivity.class);
    menuActivity.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP |
            Intent.FLAG_ACTIVITY_CLEAR_TASK |
            Intent.FLAG_ACTIVITY_NEW_TASK);
    startActivity(menuActivity);
    finish();
}
```

Figura 52 - Método `onCreate()` de la clase `LogoutActivity`

Esta actividad no tiene interfaz de usuario porque no es necesario mostrar al usuario ninguna información.

El archivo `AndroidManifest.xml` contiene las configuraciones básicas de la aplicación Android y que son necesarias para el correcto funcionamiento de la aplicación. El Anexo B, apartado 4, recoge este archivo.

## 6.3. Pruebas

Por último, si el lector desea reproducir los experimentos realizados, el Anexo A recoge una guía con todos los pasos a seguir, desde la instalación del servidor XAMPP hasta la instalación de la aplicación en un smartphone.

El testeo de la aplicación Android se ha realizado utilizando un MotoG con Android KitKat 4.4. Tanto el desarrollo como la máquina en la que se ha desplegado el servidor TFGLocation es un PC portátil con un i3 a 2.53 GHz y 8 GB de memoria RAM.

En este punto finaliza la implementación de los prototipos. Se ha creado una aplicación que envía su posición al servidor y que éste las analiza para poder definir unas reglas de comportamiento del usuario para realizar ciertas acciones sobre la casa, como pueda ser encender o apagar la climatización. En el próximo capítulo se mencionan las conclusiones alcanzadas y el trabajo futuro a realizar.

# 7. Conclusiones y trabajo futuro

En este capítulo se muestran las conclusiones que se han obtenido y las posibles ampliaciones o mejoras del trabajo realizado.

## 7.1. Conclusiones

El resultado del trabajo realizado, debido a la limitación del tiempo disponible, es el diseño del prototipo de una aplicación móvil basada en Android y del prototipo de un servidor basado en REST que analiza la posición de un habitante de una *smarthome* y gestiona la lista de la compra.

La implementación del servidor web se ha realizado con Restlet, que es un framework basado en Java para desarrollar servicios web con una capa REST y que ofrece al desarrollador la posibilidad de implementar dichos servicios de una forma muy sencilla y abstrayendo la gestión de las peticiones HTTP.

Por otro lado, la implementación de la aplicación móvil basada en Android ha supuesto el estudio de la geolocalización de dispositivos móviles, utilizando para ello los distintos servicios que proporcionan dichos dispositivos y del uso de servicios que ofrecen empresas como Google.

A nivel académico se pone de manifiesto que el aporte de conocimiento y praxis por el uso de un framework no utilizado en las asignaturas estudiadas dentro de la titulación, así como el desarrollo de un servidor web basado en REST desde cer, supone un reto en sí mismo como culmen de las nociones aprendidas en los últimos años con respecto a este tipo de servicios.

Por su parte, Android, una de las plataformas más ampliamente utilizadas, ha permitido, a través de la implementación de la aplicación, profundizar en conocimientos introducidos por experiencias académicas previas. Un ejemplo de los conocimientos que se han adquirido pueden ser la inserción y personalización de los mapas, y las APIs que proporciona Google para el geoposicionamiento.

## 7.2. Trabajo futuro

El trabajo futuro que le sigue a este proyecto es el de que el servidor implementado soporte múltiples usuarios en cuanto a la aplicación de las reglas ya que, tal y como se ha estructurado, hay una única lista con todas las distancias, con respecto de la casa, de todos los usuarios. Para ello, habría que crear una lista por cada habitante de la casa que tenga la aplicación instalada y comparar las distancias calculadas en cada momento para poder aplicar una regla u otra.

En este trabajo se han manejado sólo algunos de los dispositivos que forman parte de la domótica de la casa, así que hay muchas opciones disponibles como por ejemplo encender el equipo de música con un tipo de música u otro dependiendo de si el usuario ha estado todo el día trabajando o ha estado de compras o de excursión.

Otra de las posibles ampliaciones sería que el usuario no se tuviera que preocupar por mantener la lista de la compra actualizada, sino que se actualizara de forma automática el stock de la nevera y la despensa con cada compra y cada vez que se consumiera un producto se añadiera a la lista.

Por lo que respecta a la aplicación móvil, se podría añadir más funcionalidad: el usuario podría ver en cada momento qué dispositivos están conectados en su casa y cuáles no, y poder controlarlos desde el *smartphone*, independientemente de las reglas que se puedan aplicar según su posición.

# 8. Referencias


Amaro, J. E. (2012). *El gran libro de programación avanzada con Android.* Marcombo.

Fons Cors, J. (2014). *SmartHome Server: Infraestructura para prototipar aplicaciones sobre una Vivienda Inteligente vía REST*.

Gómez, S. (2011). *sgoliver.net.* Obtenido de Preferencias en Android: http://www.sgoliver.net/blog/?p=1731

Gómez, S. (2012). *sgoliver.net.* Obtenido de Mapas en Android (Google Maps Android API v2): http://www.sgoliver.net/blog/?p=3244

Google. (2013). *Google Developers.* Obtenido de API de Google Maps para Android: https://developers.google.com/maps/documentation/android/

Google. (2014). *Google Developers.* Obtenido de La API de Google Places: https://developers.google.com/places/training/

Google. (s.f.). *Android Developers.* Obtenido de Location Strategies: http://developer.android.com/intl/es/guide/topics/location/strategies.html

Louvel, J., Templier, T., & Boile, T. (2012). *Restlet in Action.* Manning.

Restlet. (2014). *GitHub.* Obtenido de Restlet Framework for Java: https://github.com/restlet/restlet-framework-java

Restlet. (2014). *Restlet API 2.2.2 - Java Standard Edition.* Obtenido de http://restlet.com/learn/javadocs/2.2/jse/api/

Tomás, J. (2013). *El gran libro de Android.* Marcombo.

V.A. (2012). *No-IP*. Obtenido de Free Dynamic DNS: Getting Started Guide: http://www.noip.com/support/knowledgebase/getting-started-with-no-ip-com/

V.A. (s.f.). *Stack Overflow*. Obtenido de http://stackoverflow.com/

V.A. (s.f.). *Wikipedia*. Obtenido de http://en.wikipedia.org/

*Web Server Survey.* (Agosto de 2014). Obtenido de Netcraft: http://news.netcraft.com/archives/2014/08/27/august-2014-web-server-survey.html

# Anexos

## Anexo A: Guía de instalación

En este anexo se va a describir al lector de este trabajo los pasos que debe realizar para poder reproducir este trabajo en un entorno Windows.

### 1. Instalación del servidor XAMPP

El primer paso a realizar es la instalación del servidor XAMPP. Para ello, ha de obtenerse el archivo ejecutable disponible en https://www.apachefriends.org/es/index.html. Una vez instalado, siguiendo los pasos de instalación del asistente, sólo habrá que abrir el panel de control del servidor y arrancar el servidor Apache, tal y como se muestra en la Figura 4.

### 2. Creación de un host en No-IP e instalación de DUC

Se va a utilizar No-IP, un proveedor de DDNS, para crear un nombre de host que lo asociará a una IP estática. Para ello, hay que crear una cuenta en No-IP y, una vez registrados, añadir un host. La siguiente figura muestra cómo hacerlo:

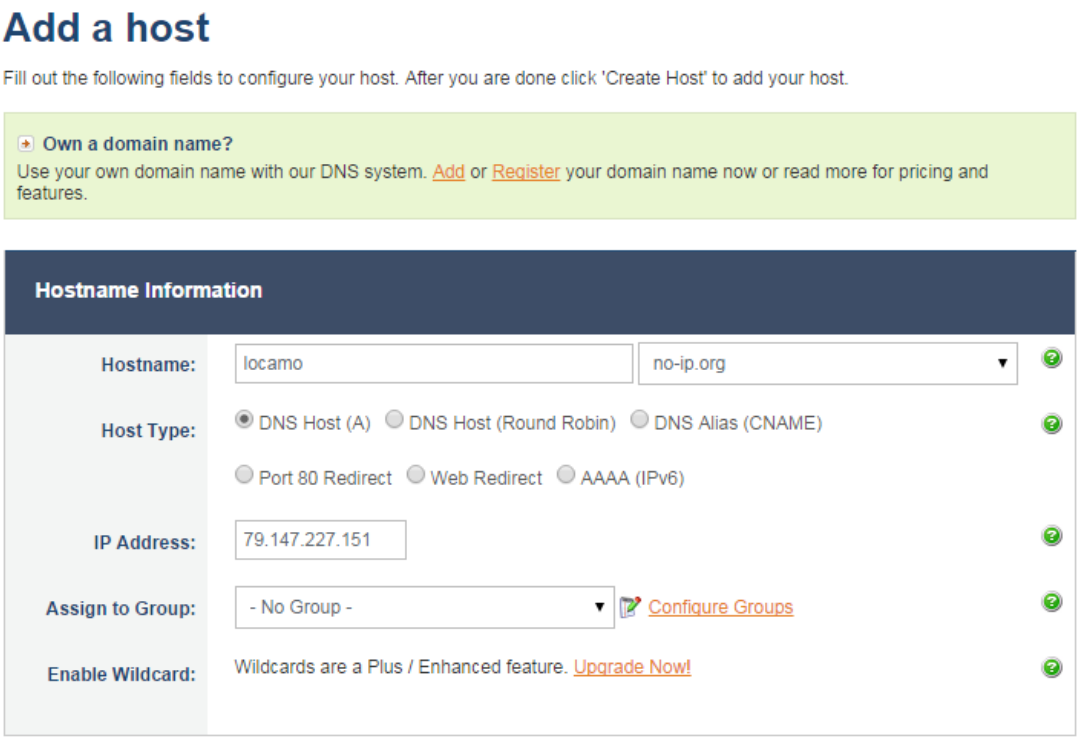



Figura 53 - Añadir un host en No-IP

Una vez creado, es recomendable bajarse el cliente DUC, que actualizará el DNS, si la IP cambia, de forma transparente al usuario. El programa se baja de esta dirección https://www.noip.com/download?page=win. Una vez instalado, hay que registrarse con el usuario que antes se ha creado. A continuación se muestran la secuencia de pasos que se deben realizar:

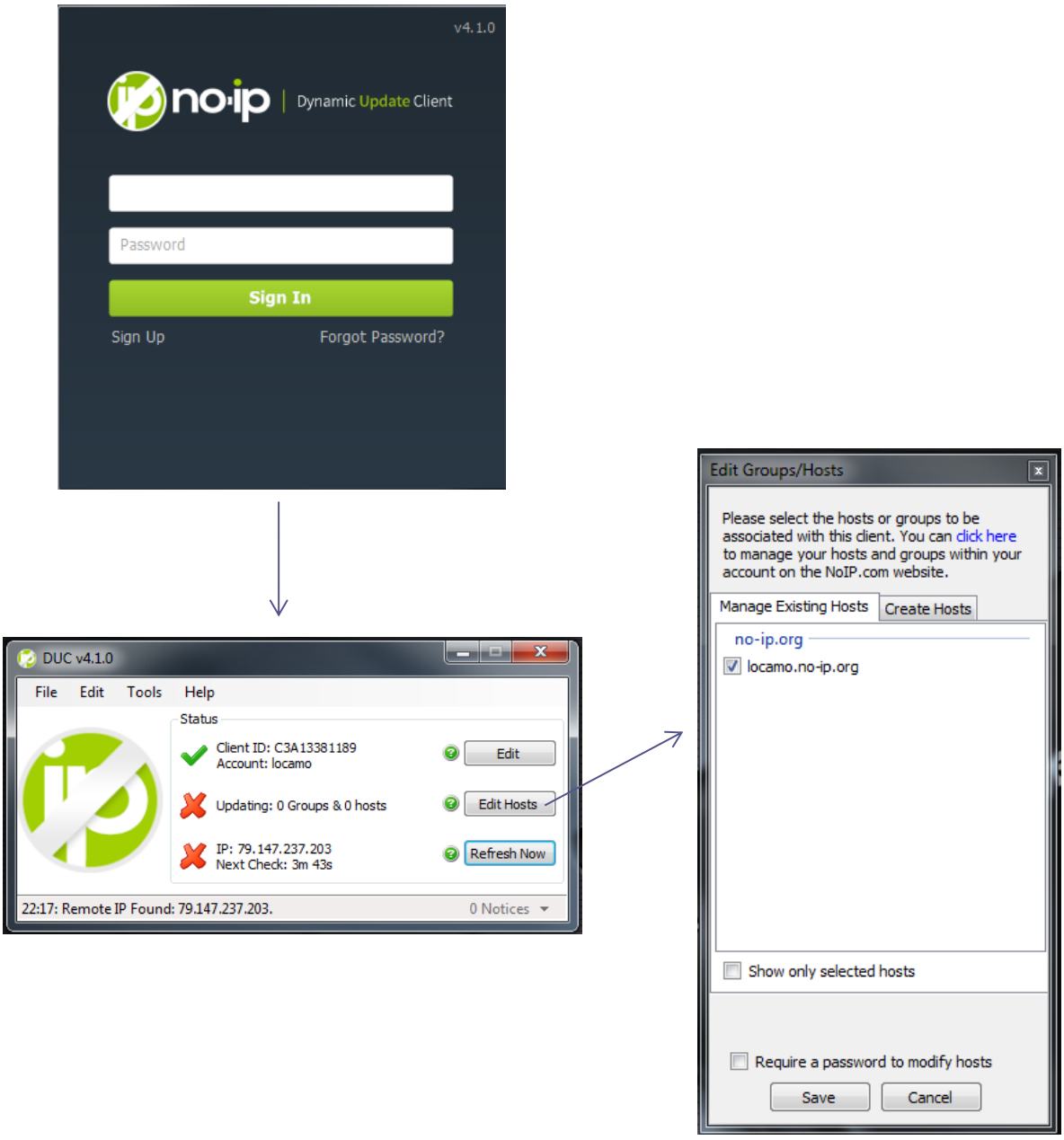



**Figura 54 - Secuencia de pasos para la configuración de DUC**

## 3. Configuración de la red

Llegados a este punto, hay que editar la configuración IPv4 de forma que la IP privada sea estática para poder abrir los puertos posteriormente en el router a esa IP.

Para ello, vamos a la red a la que estamos conectados, abrimos el menú contextual, seleccionamos "Estado" -> "Propiedades" -> "Protocolo de Internet versión 4 (TCP/IPv4)" y definimos la dirección IP que queramos, con la máscara de subred y la puerta de enlace asociadas al router. Aceptamos los cambios.

A continuación, hay que abrir los puertos (port forwarding) del router para que se pueda redireccionar las peticiones de dichos puertos a los puertos internos. Tal y como se ha mostrado durante todo el trabajo, los puertos que se tienen que abrir son 80 (servidor web), el 8182 (servidor SmartHome) y el 8284 (servidor TFGLocation).

El router sobre el que se ha realizado el port forwarding es un Comtrend, y para poder llevar a cabo esto, hay que ir a "Advanced Setup" -> "NAT" -> "Virtual Servers" y añadir uno nuevo. Para ello, hay que indicar el nombre del servidor virtual que se va a crear, la dirección IP privada asociada y los puertos. La siguiente figura muestra la configuración para abrir el puerto 8182; habría que hacer lo mismo para el resto de puertos:

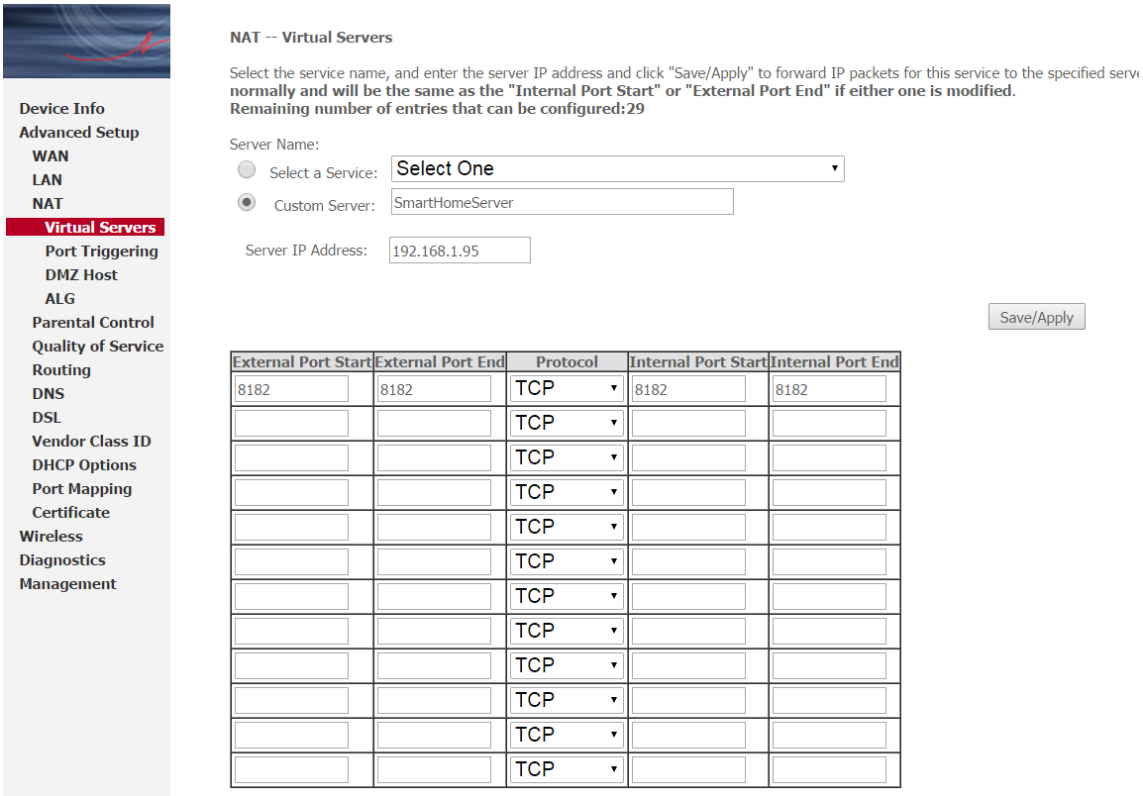



Figura 55 - Configuración para abrir un puerto en el router

### 4. Arrancar el servidor SmartHome

Para arrancar el servidor SmartHome, hay que descomprimir el fichero disponible en este enlace http://goo.gl/sc4W4u y ejecutar en el terminal el comando `java -jar org.eclipse.osgi_3.8.0.v20120529-1548.jar -console –consoleLog`.

En la carpeta que se ha descomprimido, hay un manual de usuario en el que se muestra qué servicios hay que iniciar al arrancar el servidor, las pruebas que se pueden hacer, los dispositivos que hay en la casa y su tipo y cómo realizar las pruebas.

### 5. Arrancar el servidor TFGLocation

El código del servidor TFGLocation se encuentra en el repositorio de GitHub https://github.com/locamo/tfg/tree/master/TFG-LocationServer, por lo que hay que bajarse el proyecto entero e importarlo en Eclipse. Una vez hecho esto, se ejecutará la clase `ServerComponent` y aparecerá en la consola del IDE una petición para escribir la dirección y el puerto del servidor SmartHome; al estar en la misma máquina, se pondrá `localhost 8182`.

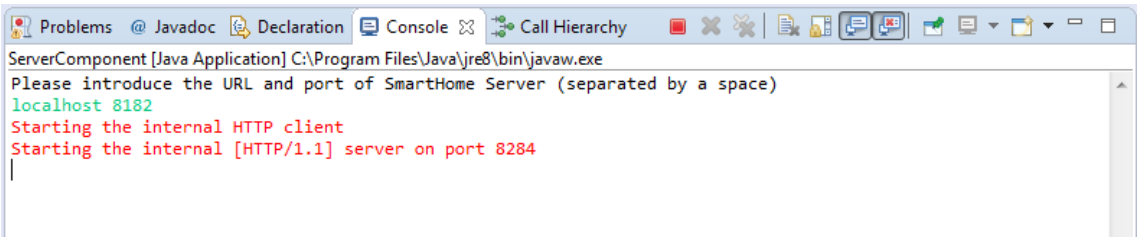



**Figura 56 - Consola de Eclipse con el servidor TFGLocation ejecutándose**

### 6. Instalar la aplicación en el Smartphone

Por último, solo queda instalar la aplicación en el dispositivo móvil. Para ello, hay que obtener el archivo .apk del siguiente enlace http://goo.gl/dCdmUi e instalarla.

Para que todo funcione correctamente, se deberá activar el GPS del dispositivo y conectar los datos móviles.

# Anexo B: Clases auxiliares

En este anexo se muestran aquellas clases que no se han expuesto en el apartado 6 de la implementación de la aplicación Android.

### Apartado 1. Clase `Utils`

La clase `Utils` se ha implementado con el fin de guardar el usuario introducido en la pantalla de registro y poder obtener el nombre de ese usuario desde cualquier clase. Para ello, se hace uso de la clase `SharedPreferences`, que gestiona los ficheros con las preferencias. Otra funcionalidad de esta clase, es la de comprobar si la conexión WiFi; se hará uso de este método en la clase `DevicePosition`.

```
public class Utils {

    private static SharedPreferences prefs;
    private static SharedPreferences.Editor editor;
    private static String user;
    private static ArrayList<String> shoppingList;

    public static String getUser(){
        return user;
    }

    public static String getUser(Context context){
        prefs = context.getSharedPreferences("tfg_preferences",
Context.MODE_PRIVATE);
        user = prefs.getString("username", "user");
        return user;
    }

    public static void setUser(Context context, String u){
        prefs = context.getSharedPreferences("tfg_preferences",
Context.MODE_PRIVATE);
        editor = prefs.edit();
        user = u;
        editor.putString("username", u);
        editor.commit();
    }

    public static void deleteCurrentUser(Context context){
        prefs = context.getSharedPreferences("tfg_preferences",
Context.MODE_PRIVATE);
        editor = prefs.edit();
        user = "user";
        editor.putString("username", "user");
        editor.commit();
    }

    public static boolean checkWifiConnection(Context context) {
        ConnectivityManager conectManager = (ConnectivityManager)
context.getSystemService(Context.CONNECTIVITY_SERVICE);
        if (conectManager != null) {
            NetworkInfo wifiConection =
conectManager.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
            if (wifiConection != null
                    && wifiConection.isAvailable()
                    && wifiConection.getDetailedState() ==
```

```
NetworkInfo.DetailedState.CONNECTED) {
                return true;
            }
        }
        return false;
    }
}
```

Figura 57 - Clase `Utils`

### Apartado 2. Clase `DevicePosition`

La clase DevicePosition es la encargada de enviar al servidor y dibujar en el mapa la posición actualizada del usuario; además, si hay elementos en la lista de la compra, también mostrará en el mapa supermercados y creará una notificación indicándolo.

La posición se actualiza cada 5 minutos para así evitar hacer demasiadas peticiones PUT al servidor TFGLocation. Esto se controla en el método `getCurrentLocation()`, que obtiene la posición del usuario ya sea por red o por GPS dependiendo de qué dispositivo esté activo, la muestra en el mapa (centrando la vista del mapa sobre dicha posición) y la envía al servidor en formato JSON (y para ello utiliza el método `createJSONLocation()`).

Además el método que obtiene la posición comprobará cada vez que se actualiza la posición si hay elementos en el servidor en la lista de la compra; si los hay, se añadirán en el mapa las localizaciones de los supermercados más cercanos en un radio de 300 metros y mostrará una notificación.

Para obtener los sitios de un determinado sitio más cercanos a una determinada posición, se ha utilizado la API de Google Places; que devolverá un array JSON con todos los sitios que ha encontrado acorde a los parámetros de búsqueda con su información asociada: nombre, localización, tipo y calificación son algunos de sus atributos.

En caso de tener elementos en la lista de la compra, además de mostrar los supermercados en el mapa, lanzará una notificación para avisar al usuario de que hay sitios cercanos a la posición en la que se encuentra donde puede realizar la compra.

```
public class DevicePosition {

    private static LocationManager locationManager;
    private static LocationListener locationListener;
    public static Location location = null;
    public static double longitude;
    public static double latitude;
    public static float accuracy;
    public static long time;
    private LatLng CURRENT_LOCATION;
    private Marker currentPositionMarker = null;
    private JSONObject jsonResult;
    private int notificationID = 1;

    private long minTime = TimeUnit.MINUTES.toMillis(5);

    // flag for GPS status
    static boolean isGPSEnabled = false;
    // flag for network status
    static boolean isNetworkEnabled = false;

    private JSONObject createJSONLocation() {
        try {
            JSONArray jsonLocation = new JSONArray();
            JSONObject jsonObjLocation = new JSONObject();
            jsonObjLocation.put("longitude", longitude);
            jsonObjLocation.put("latitude", latitude);
            jsonObjLocation.put("accuracy", accuracy);
            jsonObjLocation.put("time", time);
            jsonLocation.put(jsonObjLocation);

            JSONObject jsonResult = new JSONObject();
            jsonResult.put("location", jsonLocation);

            return jsonResult;
        } catch (JSONException e) {
            e.printStackTrace();
            return null;
        }
    }
    public void getCurrentLocation() {

        locationManager = (LocationManager)
MainActivity.getContext().getSystemService(Context.LOCATION_SERVICE);

        isGPSEnabled =
locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);
        if (Utils.checkWifiConnection(MainActivity.getContext())){
            isNetworkEnabled = true;
        }

        locationListener = new LocationListener() {
            public void onLocationChanged(Location l) {
                location = l;
                longitude = l.getLongitude();
                latitude = l.getLatitude();
                accuracy = l.getAccuracy();
                time = l.getTime();

                showLocation();
```

```
                SendPositionToServerTask sendPositionToServerTask = new
SendPositionToServerTask();
                sendPositionToServerTask.execute(createJSONLocation());

                SmarthomeShoppingList shoppingList = new SmarthomeShoppingList();
                if (shoppingList.getShoppingList().size() != 0) {
                    showSupermarkets();
                }
            }

            public void onProviderDisabled(String provider) {
            }

            public void onProviderEnabled(String provider) {
            }

            public void onStatusChanged(String provider, int status, Bundle extras) {
            }
        };

        if (isNetworkEnabled)
            locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,
minTime, 0, locationListener);
        else if (isGPSEnabled)
            locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
minTime, 0, locationListener);
        else
            Toast.makeText(MainActivity.getContext(), "Please enable GPS",
Toast.LENGTH_SHORT).show();

    }

    private class SendPositionToServerTask extends AsyncTask<JSONObject, Void, Void>
{
        @Override
        protected Void doInBackground(JSONObject... jsonObjs) {
            ClientResource resource = new ClientResource("http://locamo.no-
ip.org:8284/users/" + MainActivity.getUser());
            Representation obj = new JsonRepresentation(jsonObjs[0]);
            resource.put(obj);
            return null;
        }
    }

    private void showLocation(){
        if (location != null){
            CURRENT_LOCATION = new LatLng(location.getLatitude(),
location.getLongitude());

            CameraPosition cameraCurrentPosition = new CameraPosition.Builder()
                    .target(CURRENT_LOCATION)   // Center map at home
                    .zoom(17)                   // Zoom
                    .build();
            CameraUpdate cameraUpdateCurrentPosition =
                    CameraUpdateFactory.newCameraPosition(cameraCurrentPosition);

            MainActivity.map.animateCamera(cameraUpdateCurrentPosition);

            if (currentPositionMarker != null)
                currentPositionMarker.remove();
```

```
            currentPositionMarker = MainActivity.map.addMarker(new MarkerOptions()
                    .position(CURRENT_LOCATION)
                    .icon(BitmapDescriptorFactory.fromResource(R.drawable.position)));
        }
    }

    private void showSupermarkets(){
        GooglePlacesTask webServiceTask = new GooglePlacesTask();
        try {
            jsonResult = webServiceTask.execute().get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        try {
            JSONObject result, geometry, location;
            if (jsonResult != null) {
                JSONArray jsonArray = jsonResult.getJSONArray("results");
                for (int i = 0; i < jsonArray.length(); i++) {
                    result = (JSONObject) jsonArray.get(i);
                    geometry = result.getJSONObject("geometry");
                    location = geometry.getJSONObject("location");
                    Marker supermarket = MainActivity.map.addMarker(new
MarkerOptions()
                            .position(new LatLng(location.getDouble("lat"),
location.getDouble("lng")))
                            .title(result.getString("name"))

.icon(BitmapDescriptorFactory.fromResource(R.drawable.shopping_cart)));
                }
                createNotification();
            }

        } catch (JSONException e){
            e.printStackTrace();
        }

    }


    /**
     * AsyncTask for obtain the nearby supermarkets
     */
    public class GooglePlacesTask extends AsyncTask<Void, Void, JSONObject> {

        @Override
        protected JSONObject doInBackground(Void... params) {
            InputStream is = null;
            JSONObject jObj = null;
            String json = "";
            // Making HTTP request
            try {
              String url = "https://maps.googleapis.com/maps/api/place/search/json" +
                        "?types=grocery_or_supermarket" +
                        "&location=" + location.getLatitude() + "," +
                                       location.getLongitude() +
                        "&radius=300" +
                        "&sensor=false" +
                        "&key=AIzaSyBwHLscORtSkd_ypEQctkZGizj_rsMNe5U";
```

```
                DefaultHttpClient httpClient = new DefaultHttpClient();
                HttpGet httpGet = new HttpGet(url);
                HttpResponse httpResponse = httpClient.execute(httpGet);
                HttpEntity httpEntity = httpResponse.getEntity();
                is = httpEntity.getContent();
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            } catch (ClientProtocolException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }

            try {
                BufferedReader reader = new BufferedReader(new InputStreamReader(is,
"iso-8859-1"), 8);
                StringBuilder sb = new StringBuilder();
                String line = null;
                while ((line = reader.readLine()) != null) {
                    sb.append(line + "\n");
                }
                is.close();
                json = sb.toString();
            } catch (Exception e) {
                Log.e("Buffer Error", "Error converting result " + e.toString());
            }
            try {
                if (new JSONObject(json).getString("status").equals("OK"))
                    jObj = new JSONObject(json);
            } catch (JSONException e) {
                Log.e("JSON Parser", "Error parsing data " + e.toString());
            }
            return jObj;
        }

    }

    private void createNotification(){
        Vibrator v = (Vibrator)
MainActivity.getContext().getSystemService(Context.VIBRATOR_SERVICE);
        NotificationCompat.Builder mBuilder =
                new NotificationCompat.Builder(MainActivity.getContext())
                        .setSmallIcon(R.drawable.ic_launcher)
.setContentTitle(MainActivity.getContext().getResources().getString(R.string.app_name
))
.setContentText(MainActivity.getContext().getResources().getString(R.string.notificat
ion));
        //mBuilder.setVibrate(new long[] {300}); // Doesn't work
        v.vibrate(300);
        mBuilder.setLights(Color.WHITE, 800, 800);
        // Creates an explicit intent for an Activity in your app
        Intent resultIntent = new Intent(MainActivity.getContext(),
MainActivity.class);

        // The stack builder object will contain an artificial back stack for the
started Activity.
        // This ensures that navigating backward from the Activity leads out of
        // your application to the Home screen.
        TaskStackBuilder stackBuilder =
TaskStackBuilder.create(MainActivity.getContext());
```

```
        // Adds the back stack for the Intent (but not the Intent itself)
        stackBuilder.addParentStack(MainActivity.class);
        // Adds the Intent that starts the Activity to the top of the stack
        stackBuilder.addNextIntent(resultIntent);
        PendingIntent resultPendingIntent =
                stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);
        mBuilder.setContentIntent(resultPendingIntent);
        NotificationManager mNotificationManager = (NotificationManager)
            MainActivity.getContext().getSystemService(Context.NOTIFICATION_SERVICE);
        // mId allows you to update the notification later on.
        mNotificationManager.notify(notificationID, mBuilder.build());

    }

    public static String getTimeFormatted(long x){
        Date d = new Date(x);
        DateFormat format = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        return format.format(d);
    }

    public Location getLocation(){
        return location;
    }
}
```

Figura 58 - Clase `DevicePosition`

### Apartado 3. Clase `SmarthomeShoppingList`

La funcionalidad de la clase `SmarthomeShoppingList` es la obtener la lista de la compra del servidor y añadir y eliminar elementos de dicha lista.

De esta clase cabe destacar que se obtiene el contexto de la actividad `ShoppingListActivity` para que cuando obtenga la lista de la compra del servidor con los ítems, los "dibuje" en la interfaz correspondiente de dicha `activity`.

Por lo que respecta al insertar elementos, cada vez que se añada un ítem en el `activity` de la lista de la compra, se llamará al método `addItemToShoppingList()` de esta clase y enviará una petición PUT al servidor TFGLocation. Si se quiere eliminar un elemento, se llamará al método `deleteItemFromShoppingList()`, que enviará una petición DELETE al servidor.

```
public class SmarthomeShoppingList {

    private ArrayList<String> shoppingList;
    protected ShoppingListActivity context;

    // Constructor with the context of ShoppingListActivity
    public SmarthomeShoppingList(ShoppingListActivity context){
        this.context = context;
    }
```

```
    public SmarthomeShoppingList(){
        shoppingList = null;
    }

    public void getShoppingListFromServer(){
        ReceiveShoppingListAndDrawItTask receiveShoppingListTask = new
               ReceiveShoppingListAndDrawItTask();
        receiveShoppingListTask.execute();
    }

    private class ReceiveShoppingListAndDrawItTask extends AsyncTask<Void, Void,
Void> {
        @Override
        protected Void doInBackground(Void... args) {
            ClientResource clientResource = new ClientResource(Method.GET,
                                        "http://locamo.no-ip.org:8284/shoppingList");
            Representation rep = clientResource.get();
            final ArrayList<String> list = new ArrayList<String>();
            try {
                String result = (new JsonRepresentation(rep)).getText();
                JSONObject jsonRep = new JSONObject(result);
                JSONArray jsonArray = jsonRep.getJSONArray("shopping_list");
                if (jsonArray.length() != 0) {
                    for (int i = 0; i < jsonArray.length();i++){
                        list.add(jsonArray.get(i).toString());
                    }
                }
            } catch (JSONException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
            // Thread that paints in interface the items of list
            context.runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    if (list.size() != 0) {
                        for (String s : list)
                            context.adapter.add(s);
                    }
                }
            });

            return null;
        }
    }

    public void addItemToShoppingList(String item){
        SendItemToShoppingListTask sendItemToShoppingListTask = new
                  SendItemToShoppingListTask();
        sendItemToShoppingListTask.execute(item);
    }

    private class SendItemToShoppingListTask extends AsyncTask<String, Void, Void> {
        @Override
        protected Void doInBackground(String... args) {
            String item = args[0];
            ClientResource clientResource = new ClientResource(Method.PUT,
                                        "http://locamo.no-ip.org:8284/shoppingList");
            try {
                JSONObject json = new JSONObject();
                JSONArray jsonArray = new JSONArray();
```

```
                jsonArray.put(item);
                json.put("shopping_list", jsonArray);
                Representation representation = new JsonRepresentation(json);
                clientResource.put(representation);
            } catch (JSONException e) {
                e.printStackTrace();
            }
            return null;
        }
    }

    public void deleteItemFromShoppingList(String item){
        DeleteItemFromShoppingListTask deleteItemFromShoppingList = new
                DeleteItemFromShoppingListTask();
        deleteItemFromShoppingList.execute(item);
    }

    private class DeleteItemFromShoppingListTask extends AsyncTask<String, Void,
Void> {
        @Override
        protected Void doInBackground(String... args) {
            String item = args[0];
            ClientResource clientResource = new ClientResource(Method.DELETE,
                            "http://locamo.no-ip.org:8284/shoppingList/" + item);
            clientResource.delete();
            return null;
        }
    }

    public ArrayList<String> getShoppingList(){
        GetShoppingListTask getShoppingListTask = new GetShoppingListTask();
        try {
            shoppingList = getShoppingListTask.execute().get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        return shoppingList;
    }

    private class GetShoppingListTask extends AsyncTask<Void, Void,
ArrayList<String>> {
        @Override
        protected ArrayList<String> doInBackground(Void... args) {
            ClientResource clientResource = new ClientResource(Method.GET,
                                        "http://locamo.no-ip.org:8284/shoppingList");
            Representation rep = clientResource.get();
            ArrayList<String> list = new ArrayList<String>();
            try {
                String result = (new JsonRepresentation(rep)).getText();
                JSONObject jsonRep = new JSONObject(result);
                JSONArray jsonArray = jsonRep.getJSONArray("shopping_list");
                if (jsonArray.length() != 0) {
                    for (int i = 0; i < jsonArray.length();i++){
                        list.add(jsonArray.get(i).toString());
                    }
                }
            } catch (JSONException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
            return list;
        }
    }

}
```

Figura 59 - Clase SmarthomeShoppingList

### Apartado 4. AndroidManifest.xml

El AndroidManifest es un archivo de configuración en el que se especifican los permisos necesarios para poder realizar ciertas funciones en las actividades y se añaden las actividades de las que se va a componer la aplicación y la actividad principal, que será la encargada de iniciar la aplicación.

A continuación se van a citar para qué fin se han utilizado ciertos recursos:

- Acceso a Internet: INTERNET
- Acceso a la red: ACCESS_WIFI_STATE, ACCESS_NETWORK_STATE
- Localización: ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION
- Añadir un acceso directo: INSTALL_SHORTCUT
- Mapa: WRITE_EXTERNAL_STORAGE, READ_GSERVICES

Para poder renderizar los mapas, se ha de utilizar OpenGL ES v2. Además se ha de ha de añadir la clave obtenida en Google Console Developer. Estas son las principales funcionalidades, pero también se añaden

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

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="com.android.launcher.permission.INSTALL_SHORTCUT"
/>
    <uses-permission
android:name="com.google.android.providers.gsf.permission.READ_GSERVICES" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.VIBRATE" />
```

```
    <!-- OpenGL ES v2 to rendering maps -->
    <uses-feature
        android:glEsVersion="0x00020000"
        android:required="true" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="upv.locamo.tfg.smarthome.app.MenuActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="upv.locamo.tfg.smarthome.app.MainActivity"
            android:label="@string/app_name" >
        </activity>
        <activity
            android:name="upv.locamo.tfg.smarthome.app.AboutUsActivity"
            android:label="@string/about" >
        </activity>
        <activity
            android:name="upv.locamo.tfg.smarthome.app.LoginActivity"
            android:label="@string/title_activity_login"
            android:windowSoftInputMode="adjustResize|stateVisible" >
        </activity>
        <activity android:name="upv.locamo.tfg.smarthome.app.LogoutActivity" >
        </activity>
        <activity
            android:name="upv.locamo.tfg.smarthome.app.ShoppingListActivity"
            android:label="@string/shoppingList" >
        </activity>

        <!-- Google Maps -->
        <meta-data
            android:name="com.google.android.maps.v2.API_KEY"
            android:value="AIzaSyBOnaVVYp1eo8Yi5PkwOhR8X3oBNAKD0rQ" />

    </application>

</manifest>
```

**Figura 60 - Código del `AndroidManifest.xml`**