Trabajo de Fin de Grado

ULL Maps. Aplicación de localización de infraestructuras relacionadas con La Universidad de La Laguna.

Juan Tareq González de Chávez Pérez

La Laguna, 29 de mayo de 2017

D. **Francisco de Sande González**, con DNI nº 42.067.050-G profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

"*ULL Maps. Aplicación de localización de infraestructuras relacionadas con La Universidad de La Laguna.*"

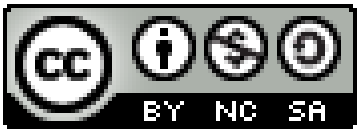
ha sido realizada bajo su dirección por D. **Juan Tareq Gonález de Chávez Pérez**, con DNI nº 79.071.887-A

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firma la presente en La Laguna a 29 de mayo de 2017

Agradecimientos

Mis agradecimientos al profesor Francisco de Sande González por su labor como tutor de este proyecto, orientando este trabajo, compartiendo su conocimiento y exigiendo siempre lo mejor.
A mis padres y amigos por apoyarme y motivarme durante el transcurso de mis estudios.

Licencia



Resumen

Este documento recopila el trabajo de investigación del alumno durante el proceso de desarrollo de una aplicación para dispositivos móviles Android mediante el uso de las API de Google Maps, Google SpreadSheet y Android Studio.

Partimos de los conocimientos de programación en Java adquiridos en la asignatura: "Programación de Aplicaciones Interactivas" cursada en el itinerario de Ingeniería de Computación y también los conocimientos adquiridos en Android en la asignatura: "Interfaces Inteligentes". Estas asignaturas, impartidas en el tercer y cuarto curso del Grado en Ingeniería Informática de "La Universidad de La Laguna", han sido las que han sentado los fundamentos a partir de los cuales se ha desarrollado el trabajo.

Durante este proyecto, el alumno ha conseguido adquirir independencia en su trabajo, visión y planificación realizando tareas de investigación, desarrollo y documentación, que han dado como resultado la obtención de conocimientos durante el proceso de trabajo.

Palabras clave: Aplicaciones Android, Java, dispositivos móviles, programación, Google Maps, Google Sheet.

Abstract

The aim of the project has been the development of an application for Android devices using the Google Maps API, Google SpreadSheet, and Android Studio.

Based on the knowledge of *Java* programming obtained in the subject: *"Architectural Design and Patterns"* and also the knowledge acquired in *Android* in the subject: *" Intelligent Interfaces"* studied in the third and fourth years of the degree in Computer Engineering from *"La Universidad de La Laguna"*. have been the foundations from which the application has been developed.

Moreover, the student has learned independence in his work and gained vision and scheduling aptitudes, developing different labors of research, development and documentation that have come to give her a wide knowledge during the development of this project.

Keywords: *Application for Android, Java, mobile devices, programming, Google Maps, Google Sheet.*

Índice general

Introducción **1**

1. Objetivos **2**

2. Herramientas y Tecnologías **3**
2.1. Herramientas de Desarrollo 3
2.1.1. Android Studio 3
2.1.2. LaTeX 4
2.1.3. GitLab 4
2.2. APIs de Google 4
2.2.1. Google Maps 5
2.2.2. Google Sheet 5

3. Aplicación ULL-Maps **8**
3.1. Especificación de requisitos 8
3.1.1. Explicación detallada 9
3.2. Ventanas de la aplicación 11

4. Configuración inicial y la información de la app **14**
4.1. Primeros pasos 14
4.2. La hoja de cálculo 15

5. Ventana inicial y manejo de la información **18**
5.1. La ventana inicial 18
5.1.1. El layout 18
5.1.2. Obtención y entrega de datos 20
5.1.3. Selección de idioma 20

6. Configuración común de los mapas **24**
6.1. InfoWindow personalizadas 24
6.2. Diseño del menú superior 27
6.3. Control de las opciones del menú superior 32

6.4. MarkersData 33

7. Ventana *FullInfo* y los mapas **35**
7.1. Ventana *FullInfo* 35
7.2. Mapas 39
7.2.1. Mapa con marcadores 40
7.2.2. Mapa con iconos 41
7.2.3. Mapa con polígonos 45

8. Conclusiones y líneas de trabajo futuras **49**
8.1. Conclusiones 49
8.2. Conclusions 50

9. Presupuesto **51**

Bibliografía **53**

Índice de figuras

2.1. Android Studio, un IDE flexible e intuitivo. 4
2.2. GoogleMaps. 6
2.3. Google Sheets 6

3.1. Ventana principal de la app `ULL-Maps`. 11
3.2. Tipos de mapeado 12
3.3. Formas de mostrar información en la app. 13

4.1. Selección de Activity inicial. 14

Introducción

Este documento comprende el trabajo de investigación y desarrollo realizado por el alumno en la consecución de su Trabajo de Fin de Grado (TFG), con el que culminará sus estudios del Grado en Ingeniería Informática cursados en la Escuela Superior de Ingeniería y Tecnología de La Universidad de la Laguna (ULL).

Capítulo 1

Objetivos

Este Trabajo de Fin de Grado tiene los siguientes objetivos principales:

- Por un lado se pretende ampliar los conocimientos en el desarrollo de aplicaciones para móviles en el sistema operativo Android [1] utilizando distintas APIs para ello.
- Teniendo en cuenta el objetivo anterior, también se pretende que se desarrollen habilidades para investigar distintas APIs y que sea posible combinar el uso de las mismas en una aplicación.
- Por otro lado, también se busca que el alumno se familiarice con el uso de herramientas de control de versiones utilizando *Gitlab* [2] y de edición de textos técnicos utilizando *LaTeX* [3].
- Por último se espera que utilizando los conocimientos adquiridos durante el TFG, el alumno sea capaz de desarrollar una aplicación que cubra los puntos mencionados.

Capítulo 2

Herramientas y Tecnologías

Este capítulo tiene como objetivo presentar las distintas herramientas software y tecnologías empleadas por el alumno en el desarrollo de este TFG.

2.1. Herramientas de Desarrollo

A continuación se explicarán brevemente las distintas herramientas software utilizadas en el proyecto.

2.1.1. Android Studio

Android Studio [4] es el IDE (Entorno de Desarrollo Integrado) oficial para el desarrollo de aplicaciones en Android, basado en IntelliJ IDEA [5]. Android Studio ofrece una serie de funcionalidades que han facilitado el desarrollar numerosas tareas, entre las cuales podemos destacar:

- Un sistema de compilación basado en Gradle [6] que ha simplificado las tareas de compilación, empaquetado y despliegue de la aplicación.
- Un emulador de dispositivo Android que permite probar la aplicación sin necesidad de instalarla en un dispositivo, aunque se ha preferido ir testeando la aplicación en un dispositivo físico.
- La posibilidad de aplicar cambios durante la ejecución de la app, ahorrando tiempo al no tener que reinstalar la app cada vez que se hace un cambio pequeño.
- Un sistema de depuración, con una interfaz sencilla e intuitiva.

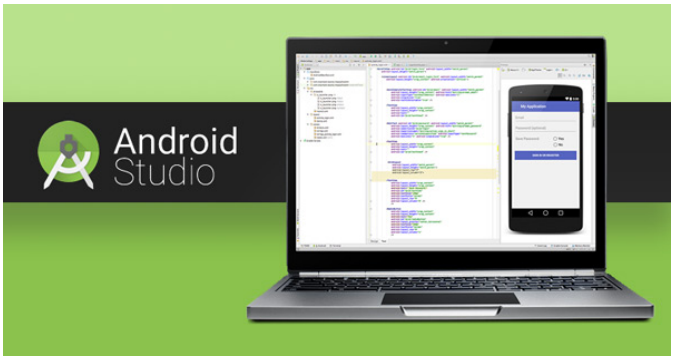


Figura 2.1: Android Studio, un IDE flexible e intuitivo.

2.1.2. LaTeX

LaTex [3] es un sistema de composición de textos que aporta una alta calidad tipográfica; el mismo está formado por un conjunto de macros escritos en 1984 por Leslie Lamport, con el objetivo de facilitar el uso del lenguaje de composición tipográfica TeX. La principal característica de LaTeX es que a diferencia de otros procesadores de texto en los cuales a medida que creamos el contenido estamos viendo el resultado final, para obtener un documento final en LaTeX hay que realizar normalmente dos etapas:

- Primero se crea un archivo en el que se almacenan las órdenes y comandos necesarios junto al texto que se quiere producir.
- Hay que procesar ese archivo y compilarlo para generar el documento final.

2.1.3. GitLab

GitLab [2] es una plataforma de desarrollo colaborativo que permite alojar proyectos bajo el sistema de control de versiones GIT. GitLab utiliza una licencia open source, siendo gratuito su uso. Se ha decidido crear un repositorio en esta plataforma, por el soporte que ofrece a la hora de llevar un control de versiones de forma privada, teniendo acceso al mismo tanto el alumno como el tutor para poder trabajar de forma conjunta llevando al día los avances del TFG. También, mediante el uso de este repositorio, se ha conseguido aumentar los conocimientos del alumno en Git, complementando los adquiridos durante la carrera.

2.2. APIs de Google

En esta sección comentaremos las diferentes APIs de Google que se han utilizado en el desarrollo del trabajo.

2.2.1. Google Maps

La API de Google Maps [7] fue publicada para Android en 2008 y en 2012 para IOS. Esta API permite utilizar mapas basados en datos de Google Maps en una aplicación Android, y además ofrece métodos para personalizar el mapa:

- Creación de marcadores, polígonos y superposiciones sobre el mapa para resaltar puntos o zonas.
- Permite cambiar la vista del usuario de modo que se muestre un área del mapa en particular.
- Ofrece la posibilidad de elegir el tipo de mapa:
  - De carreteras: muestra la vista del mapa de carreteras. Este es el modo de mapa predeterminado.
  - Satélite: muestra imágenes satelitales de Google Earth.
  - Híbrido: muestra una combinación de vistas normales y satelitales.
  - Terreno: muestra una vista basada en la información terrestre.

Aprovechando las funcionalidades de la API se crean los distintos mapas para nuestra app, utilizando para ello los objetos que nos ofrece como son los siguientes:

- Marcadores que se ubican en el mapa a través de una latitud y una longitud, los cuales además se pueden personalizar con imágenes, pudiendo clasificarlos a simple vista.
- Polígonos que permiten dibujar en el mapa las siluetas de los edificios y asignarles un evento al seleccionarlos.
- InfoWindows o ventanas de información asociadas a los marcadores y permiten asignar a cada marcador un título y una descripción del edificio o zona que representan.

2.2.2. Google Sheet

La API Google Sheet [8] es una aplicación en línea que permite a los usuarios crear y formatear las hojas de cálculo de Google. Esta API permite realizar las siguientes labores sobre una hoja de cálculo, como la que se puede observar en la Figura 2.3a:

- Importa, exporta y da formato a las hojas de datos.

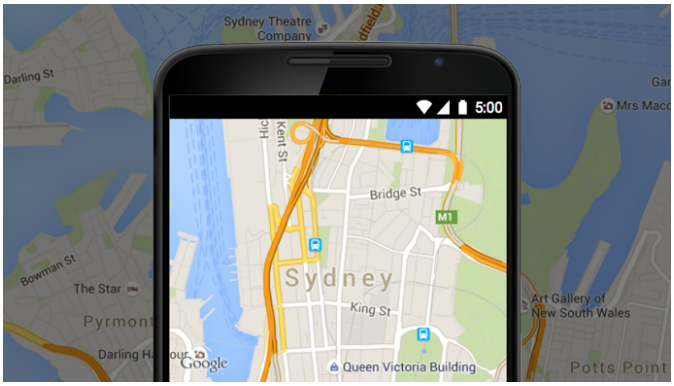
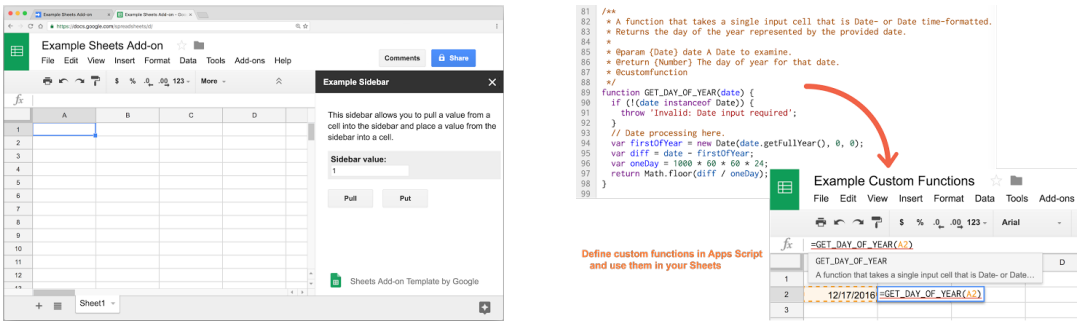


Figura 2.2: GoogleMaps.



(a) Hoja de cálculo (b) Función personalizada

Figura 2.3: Google Sheets

- Controlar el formato condicional.
- Construir y editar gráficos incrustados en una hoja.
- Configurar la validación de datos.
- Crear y actualizar intervalos con nombre y protegidos.
- Agregar y ajustar vistas filtradas.
- Crear y manipular tablas dinámicas.

Además permite definir fórmulas personalizadas (Vease Figura 2.3b) que funcionen en nuestra hoja de cálculo, aunque esta parte pertenece a la API para aplicaciones web y no a la de Android.

A la hora de acceder a la información, Google Sheets utiliza la notación A1 para saber sobre qué rango de datos se trabaja, con ella se indica de qué hoja se

desea leer o escribir la información y sobre qué rango de valores se desea trabajar. Ejemplos de rangos utilizando este sistema de notificación serían:

- Sheet1!A1:B2 se refiere a las primeras dos celdas en las primeras dos filas de la hoja Sheet1.
- Sheet1!A:A se refiere a todas las celdas en la primera columna de la hoja Sheet1.
- Sheet1!1:2 se refiere a todas las celdas en las primeras dos filas de la hoja Sheet1.
- Sheet1!A5:A se refiere a todas las celdas de la primera columna de la hoja Sheet1, desde la quinta fila hasta el final.
- A1:B2 se refiere a las dos primeras celdas en las dos primeras filas de la hoja Sheet1.
- Sheet1 se refiere a todos los datos de la hoja Sheet1.

Capítulo 3

Aplicación ULL-Maps

En este capítulo explicaremos la aplicación `ULL-Maps` utilizando la especificación de requisitos de la misma y explicando su funcionamiento.

3.1. Especificación de requisitos

A continuación se muestra la especificación de requisitos que se propuso para el desarrollo de la app objeto de este TFG.

Se trata de una aplicación para dispositivos móviles, la cual se denomina `ULL-Maps`, que permitirá al usuario localizar distintos edificios pertenecientes o con relación a la Universidad de La Laguna (ULL) en un mapa, junto a la información de los mismos.

Para cada edificio tendremos la siguiente información como base:

- Una breve descripción de las actividades que se desarrollan en el mismo y los servicios que ofrece.
- El nombre del edificio.
- Una imagen para usar como referencia del edificio.

Esta información se le mostrará al usuario según vaya seleccionando en la app un edificio, para lo que dispondrá de tres formatos distintos:

- Un mapa con marcadores.
- Un mapa con iconos.
- Un mapa con polígonos que resaltan la silueta de los edificios.

Además de esto existirá un menú que permitirá al usuario visitar distintas páginas web que tengan relación con el edificio que haya seleccionado.

Uno de los posibles usos de la app es el facilitar a nuevos estudiantes sus comienzos en la ULL, mostrándoles información de dónde encontrar:

- Las facultades.
- Las bibliotecas.
- Los aularios.
- Salas de estudio.
- Otros recursos de la universidad.

Todo esto directamente en su móvil. También podrá ser de ayuda a cualquier persona que visite la ULL, facilitando su visita al tener localizados los distintos edificios de la ULL. Además, la aplicación contará con tres idiomas disponibles:

- Español.
- Inglés.
- Alemán.

Haciendo más sencillo su uso para usuarios extranjeros.

3.1.1. Explicación detallada

La aplicación `ULL-Maps` comenzará mostrando una pantalla de inicio en la cual podremos elegir entre los distintos tipos de mapa disponibles:

- Mapa con polígonos, en el cual se resalta la silueta de los edificios.
- Mapa con marcadores, en el que tendremos un punto en el mapa por cada edificio. Al pulsar sobre uno de ellos se mostrará información sobre el edificio seleccionado.
- Mapa con iconos, los iconos tendrán un funcionamiento similar a los marcadores, con la diferencia de que para cada tipo de edificio mostrará un icono diferente que represente la función del mismo.

En su primer uso, la aplicación solicitará al usuario permisos para acceder a una cuenta de Google y le indicará que se necesita de conexión a Internet si no dispone de ella, ya que la aplicación requerirá de estos servicios para su correcto funcionamiento.

En la ventana inicial de la misma, se pueden realizar dos acciones:

- Cambiar el idioma, para lo que se dispone de botones representados por banderas.
- Seleccionar un tipo de mapa para la aplicación.

Al seleccionar un tipo de mapa, se procederá a cargar aquel que el usuario haya solicitado. Para cada tipo de mapeado necesitaremos información, que obtendremos de una hoja de cálculo, la cual procederemos a explicar a continuación:

- Polígonos: Requieren de varios puntos del mapa, los cuales deberán ser almacenados por orden de sus vértices para luego poder dibujarlos en el mapa.
- Marcadores: Requerirán de un punto en el mapa, localizado sobre el edificio al que representan.
- Iconos: Utilizarán la misma referencia que los marcadores y además necesitarán de una imagen, la cual está incluida en la aplicación.
- Común: Todos los tipos de mapas tendrán información en común para cada edificio, como la descripción del mismo y el tipo de edificio que es.

Para obtener estos datos se utilizan las hojas de cálculo de Google, donde estará guardada la información de cada edificio, siendo obligatorio tener al menos la información común y un punto central, representado con una latitud y longitud, para poder representarlos correctamente.

De cada edificio dispondremos de los siguientes datos:

- Descripción de las actividades y/o estudios que se realicen en ellos.
- Imagen para poder identificarlos una vez se llega al destino.
- El nombre del edificio.
- El tipo de edificio del que se trata.
- Sus datos de ubicación para poder situar al edificio, un punto en el caso de iconos y marcadores y una serie de puntos para los polígonos.

Al seleccionar un edificio en el mapa, además de mostrar la descripción del mismo se podrá disponer de información adicional en un menú desplegable para el usuario, el cual estará situado en la parte superior derecha. En este menú dispondremos de distintas URL asociadas al edificio, pudiendo seleccionarlas para su visualización en el navegador del dispositivo. Este menú se actualiza cada vez que se seleccione un edificio del mapa. También, se puede ampliar

la información disponible en la *InfoWindow* del edificio clicando sobre esta, lo que abrirá en la aplicación una nueva ventana que contiene toda la información disponible y permitiendo al usuario llamar o enviar un e-mail, si el edificio dispone de ellos, seleccionando el número o el correo.

Dentro de la aplicación, tras pasar el menú inicial y haber seleccionado un tipo de mapa, podremos cambiar el tipo de mapa utilizando el botón de opciones del dispositivo o seleccionando en el menú superior la opción *Mapa*, el cual, al ser presionado, nos dará a elegir qué estilo queremos visualizar: polígonos, marcadores o iconos.

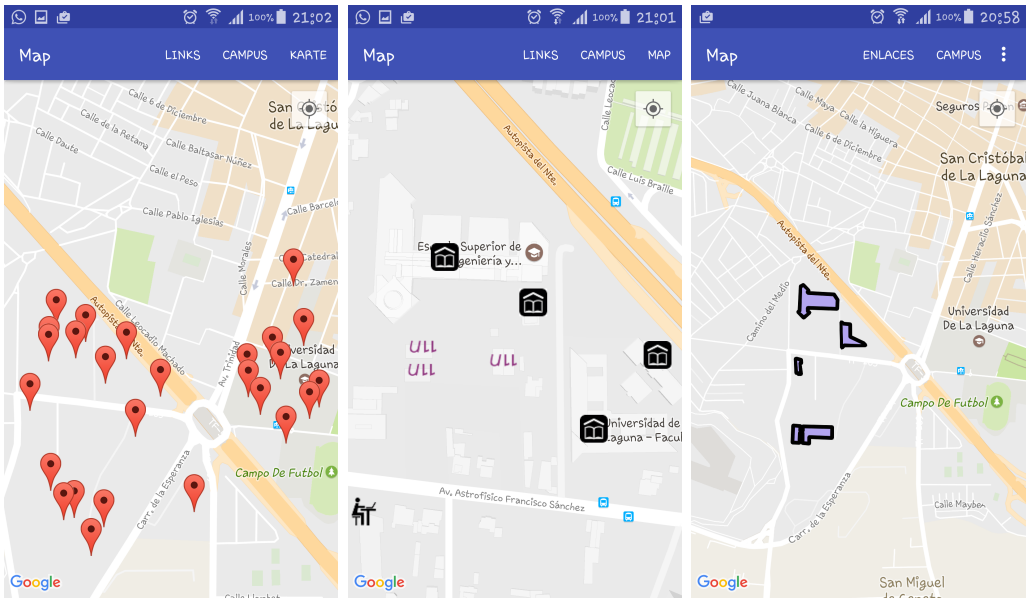
3.2. Ventanas de la aplicación

`ULL-Maps` comienza con una ventana inicial (Vease la Figura 3.1) en la que se puede realizar un cambio de idioma, utilizando las banderas que se encuentran en esta, o ir a uno de los tres tipos de mapeado, por medio de los botones que se encuentran en la parte inferior.

Figura 3.1: Ventana principal de la app `ULL-Maps`.

Al seleccionar uno de los botones, se procede a cambiar la pantalla de la app, mostrando el mapa seleccionado al usuario. Ahora se procederá a explicar cómo se muestra cada tipo, acompañado de una imagen:

- Mapa con marcadores (Vease la Figura 3.2a): estará compuesto por marcadores simples, representando cada uno de ellos a un edificio.



(a) Marcadores (b) Iconos (c) Polígonos

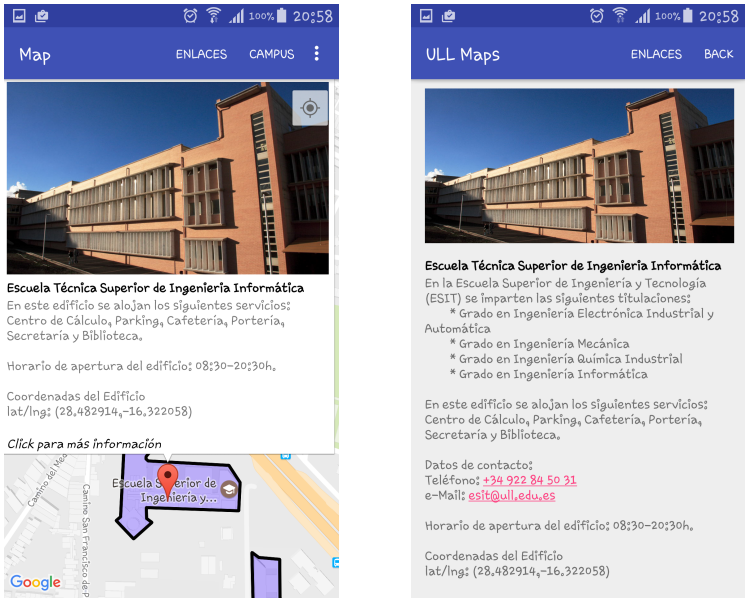
Figura 3.2: Tipos de mapeado

- Mapa con iconos (Vease la Figura 3.2b): a diferencia de los marcadores simples en este mapeado se muestran diversos iconos, según de qué tipo sea el edificio al que representan.
- Mapa con polígonos (Vease la Figura 3.2c): resalta la silueta de los edificios utilizando polígonos.

Sin importar el tipo de mapeado hay algunas funciones que se comparten entre ellos:

- Todos constan de un menú superior el cual se explica en la sección 6.2 del capítulo 6.
- En cada uno de ellos, al seleccionar un marcador, icono o polígono, se muestra una ventana de información cuyo contenido depende del edificio al que represente, se puede observar esta ventana en la Figura 3.3a.

Teniendo una *InfoWindow* abierta, se puede clicar sobre ésta para ir a una nueva ventana, en la que se muestra información más detallada del edificio seleccionado, que aquella mostrada en la pantalla del mapa. Se puede observar esta ventana en la Figura 3.3b.



(a) *InfoWindow* para el mapa con polígonos.

(b) Ventana de información detallada.

Figura 3.3: Formas de mostrar información en la app.

Capítulo 4

Configuración inicial y la información de la app

Teniendo como base lo expuesto en el capítulo anterior, éste se dedicará a explicar los pasos que se han seguido en el desarrollo de `ULL-Maps`.

4.1. Primeros pasos

Lo primero que se necesita para elaborar la aplicación es su esqueleto. Para hacerlo, utilizando Android Studio [4], bastará con crear un nuevo proyecto y seguir los pasos que nos indica el IDE. Cuando el IDE pregunte si se quiere elegir una Activity inicial, solicitar la de Google Maps Activity (Vease la Figura 4.1), con ello el IDE creará el esqueleto necesario para desarrollar una app que muestre un mapa.

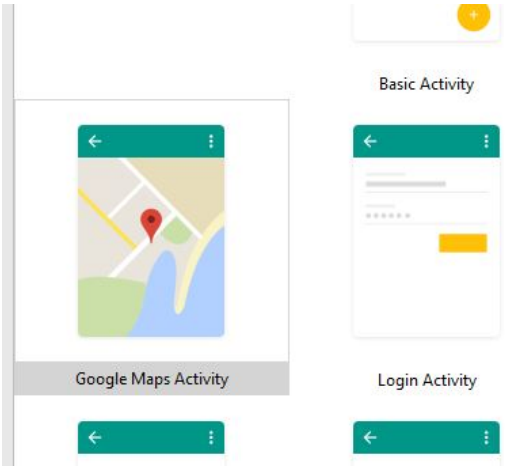


Figura 4.1: Selección de Activity inicial.

Aún teniendo el esqueleto de la aplicación, por defecto no se podrá ver el mapa de Google. Para poder utilizarlo se tiene que activar en la consola para desarrolladores de Google [9], una vez situado en ella se crea un nuevo proyecto. Una vez creado se tendrá que habilitar para ese proyecto las API que se desee utilizar, que en nuestro caso serán dos:

- Por un lado se tiene la API de Google Maps Android [7]. Al habilitarla se nos dará una clave de API la cual se tiene que registrar en la aplicación, en el fichero *app->res->values->google_maps_api.xml*, una vez hecho esto ya se podrá visualizar el mapa en la app. Esta clave será válida para todas las actividades de la aplicación que utilicen un mapa.

- Por otro lado está Sheets API [8]. Ésta solicitará crear un ID de cliente de OAuth 2.0. Para ello se necesita una huella digital de certificado de firma, que se obtiene ejecutando el siguiente comando

```
1 $ keytool -exportcert -alias androiddebugkey -keystore ~.android/debug.keystore -list -v
```

  En caso de que la consola no ejecute el comando, se puede encontrar el *keytool* en la carpeta de *Java->Jre->bin*. Una vez hecho se podrá obtener la clave SHA1 que solicita la consola de Google, esto, junto al nombre del paquete de la aplicación, creará un ID de cliente OAuth 2.0 y con ellos se podrá utilizar la API en la aplicación.

Una vez realizados estos procesos se podrá comenzar a trabajar en el código de la aplicación, teniendo disponibles las API.

4.2. La hoja de cálculo

Para poder dibujar en el mapa de Google los distintos objetos que definan la app, se necesita de cierta información, la cual, como se ha comentado con anterioridad se implementa mediante hojas de cálculo en la nube. En este apartado se explicará cómo está distribuida la información.

Lo primero que hay que tener en cuenta es qué información básica se necesita. Analizando la API de Google Maps [7], se sabe que datos puede guardar el objeto *Marker* que son:

- Un título.

- Una descripción.

- Sus coordenadas.

Además de los datos que puede contener, tiene varias opciones que permiten personalizar el marcador, como por ejemplo poner iconos en lugar de la imagen por defecto.

En base a esta información se puede deducir que se necesitan casillas para cada uno de estos datos. Dos en el caso de las coordenadas, una para la latitud y otra para la longitud. También es necesario clasificar los edificios según su campus y qué actividad se realiza dentro de estos. Las tipologías de edificio definidas en la app son:

- Facultad.
- Instituto.
- Zona de estudio.
- Aulario.
- Biblioteca.
- Servicios varios.

Y para terminar, se desea tener información de distintas páginas relacionadas con el edificio, alguna imagen del mismo, para poder localizarlo con la vista y coordenadas varias para poder dibujar su silueta en el mapa utilizando los *Polygons* que ofrece la API. Por lo que en total se requieren ocho celdas fijas y un número no determinado de celdas que representan los vértices del edificio en el mapa. Sabiendo la información que se necesita, se distribuye en el siguiente orden en la hoja de cálculo que utiliza la aplicación `ULL-Maps`:

1. Campus al que pertenece el edificio.
2. Nombre del edificio.
3. Actividad que se realiza.
4. Latitud.
5. Longitud.
6. Descripción.
7. Distintas URL asociadas, incluyen un nombre representativo (Nombre-URL) para que en la aplicación el usuario no tenga que adivinar a dónde le lleva la URL.
8. URL que contenga una imagen del edificio.

9. En adelante serán distintas coordenadas, latitud y longitud separadas por coma, que representan los vértices del polígono que representa al edificio, en caso de que este disponible.

En cuanto a la seguridad de la información, ésta se define en la propia hoja de cálculo, pudiendo cualquier usuario leer datos de la misma pero no escribir en ella. Las personas que pueden escribir en la hoja, al menos durante la elaboración de este TFG, son el alumno y el tutor, aunque en el futuro se puede modificar estos permisos.

Capítulo 5

Ventana inicial y manejo de la información

En este capítulo se procederá a explicar cómo se define la ventana principal de la app, dónde y cómo esta registrada la información que utiliza y los métodos utilizados para extraer y comunicar los datos obtenidos entre las distintas ventanas de la aplicación.

5.1. La ventana inicial

La ventana inicial de `ULL-Maps` es en la que se realiza tanto la obtención de los datos guardados en la hoja de cálculo que contiene la información que necesita la app, como la selección de idioma. Puede verse la hoja utilizada para `ULL-Maps`en la siguiente referencia [**?**]. En esta sección se explica cómo se realizan esas tareas.

5.1.1. El layout

Antes de proceder a explicar las tareas que se pueden realizar en esta ventana de la app, hay que definir su layout, es decir el archivo XML que la define, el cual se puede ver en el Listado 5.1.

```
1  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
2      xmlns:tools="http://schemas.android.com/tools"
3      android:orientation="vertical"
4      android:background="@drawable/ullmapsbg"
5      ... >
6      <Button
7          android:text="@string/mapa_marcadores"
8          android:layout_width="match_parent"
9          android:layout_height="wrap_content"
10         android:id="@+id/button2"
11         style="@android:style/Widget.DeviceDefault.Button.Inset"
12         android:onClick="mapaMarkers_b"
13         android:layout_alignParentBottom="true"
14         android:layout_alignParentStart="true" />
15
16     <Button
17         android:text="@string/mapa_iconos"
18         ... />
19     <Button
20         android:text="@string/mapa_poligonos"
21         ... />
22     <Button
23         android:id="@+id/en"
24         android:tag="en"
25         style="@style/Widget.AppCompat.Button"
26         android:layout_width="wrap_content"
27         android:layout_height="wrap_content"
28         android:background="@drawable/banderaing"
29         android:onClick="changeLanguage"
30         ... />
31     <Button
32         android:id="@+id/es"
33         android:tag="es"
34         ... />
35     <Button
36         android:id="@+id/de"
37         android:tag="de"
38         ... />
39 </RelativeLayout>
```

Listado 5.1: Fichero XML que define de la ventana inicial de la aplicación.

En este fichero se define el fondo que debe tener, el cual es una imagen propia de la aplicación y seis botones, los cuales están divididos en dos grupos, que tienen las siguientes funciones:

- Seleccionar el tipo de mapa que se desea. Estos tres botones, uno por cada tipo de mapa, tienen registrado, cada uno, un método al seleccionarlos, que se encargará de cambiar la ventana actual del usuario. Se explica más detalladamente en la subsección 5.1.2.

- Cambiar el idioma en el que trabaja la aplicación. Estos botones son los tres últimos que se definen en el archivo XML 5.1 (líneas 22-38) y además de que, a diferencia de los botones que permiten cambiar de mapeado, invocan al método *changelanguage()* al ser seleccionados. Para saber qué idioma

ha seleccionado el usuario, éstos tienen una etiqueta, que contiene las siglas del nuevo lenguaje que debe utilizar la aplicación. Se explicará más detalladamente cómo funciona en la subsección 5.1.3.

5.1.2. Obtención y entrega de datos

Ahora se explicará cómo se recogen los datos que utiliza la app. Puesto que este proceso se realiza sólo en la apertura de la aplicación, la recogida de datos se hará asociada con la pantalla inicial de la app, y ya que al crear el proyecto se eligió que fuera un mapa, se tiene que crear una nueva Activity, en este caso vacía. Para ello desde el IDE de Android Studio [4] se selecciona la carpeta del proyecto y luego sobre *new->activity->empty activity* , creando con ello un layout y su archivo Java asociado. Ahora que se tiene la Activity, hay que situarse en su fichero Java asociado y agregar la clase privada encargada de realizar una petición al SpreadSheet y obtener su información.

Al crear un objeto de la clase *MakeRequestTask* se tendrá que dar también una credencial de una cuenta Google. Con ella creará un servicio que será el que utilice la API Sheets, por lo que la cuenta tendrá que tener, al menos, permisos de lectura sobre el SpreadSheet representado por el *spreadsheetid* que se puede ver en el método *getDataFromApi()* que abarca las líneas de la 25 a la 37 del Listado 5.2. Si todo funciona correctamente se accederá a la información y se podrá guardar esa información en una lista de listas de objetos.

Hecho esto ya se tendrá toda la información guardada en una variable y lo único que faltaría es enviar esta información al resto de actividades de la aplicación, para lo cual sólo se tendrá que pasar la lista como un extra según se cambie de actividad. Para ello se utilizan los métodos, que se deberán ejecutar según qué opción de la pantalla principal se seleccione, que se pueden ver en el Listado 5.3.

Con esto se tiene cubierta la obtención de datos y el cómo comunicarlos a las distintas actividades de la aplicación.

5.1.3. Selección de idioma

La aplicación funciona en tres idiomas, como se indicó en la especificación de requisitos, por lo que es necesario darle al usuario la opción de cambiar el idioma en el que quiera usar la aplicación. Para ello se muestran tres banderas en la ventana principal, como se puede ver en la Figura 3.1, que permiten cambiarlo.

Estas banderas realmente son botones que utilizan como fondo una imagen, se puede ver en el fichero XML del Listado 5.1 (línea 29) cual es el método que ejecutan estos al ser seleccionados y que tienen una etiqueta. Ahora se explicará qué ocurre cuando se ejecuta:

```
private class MakeRequestTask extends AsyncTask<Void, Void, List<List<Object>>> {
    private com.google.api.services.sheets.v4.Sheets mService = null;
    private Exception mLastError = null;

    MakeRequestTask(GoogleAccountCredential credential) {
        HttpTransport transport = AndroidHttp.newCompatibleTransport();
        JsonFactory jsonFactory = JacksonFactory.getDefaultInstance();
        mService = new com.google.api.services.sheets.v4.Sheets.Builder(
                transport, jsonFactory, credential)
               .setApplicationName("ULL Maps")
                .build();
   }
    //Llama a la Google Sheets API.
    @Override
    protected List<List<Object>> doInBackground(Void... params) {
        try {
            return getDataFromApi();
        } catch (Exception e) {
            mLastError = e;
            cancel(true);
            return null;
        }
    }
    // Devuelve una lista con los datos del spreadsheet
    private List<List<Object>> getDataFromApi() throws IOException {
        //Indicamos el identificador del SpreadSheet que utilizaremos
        String spreadsheetId = "1f91M3ZAzz5uYrOSs4UfZOSx3B1fqQO7HXoTzlwAaaD8";
        //En range introducimos el nombre de la pagina para obtener todos los datos.
        String range = "UllMaps";

        ValueRange response = this.mService.spreadsheets().values()
                .get(spreadsheetId, range)
                .execute();
        List<List<Object>> values = response.getValues();
        Datos = values;
        return values;
    }

    @Override
    protected void onPreExecute() {
        mProgress.show();
    }

    @Override
    protected void onPostExecute(List<List<Object>> output) {
        mProgress.dismiss();
    }
}
```

Listado 5.2: Esta clase contiene el código necesario para solicitar la información a la nube.

```
1  /* Metodos encargados de cambiar de pantalla para cada boton */
2
3  public void mapaMarkers_b(View view) {
4      Intent intent = new Intent(this, MapsActivity.class);
5      //Aqui es donde le anadimos a la actividad los datos del spreadsheet.
6      intent.putExtra("Datos", (Serializable) Datos);
7      startActivity(intent);
8  }
9
10 public void mapaIcons_b(View view) {
11     Intent intent = new Intent(this, MapsActivity_Icons.class);
12     intent.putExtra("Datos", (Serializable) Datos);
13     startActivity(intent);
14 }
15
16 public void mapaPoligonos_b(View view) {
17     Intent intent = new Intent(this, MapsActivity_Poligons.class);
18     intent.putExtra("Datos", (Serializable) Datos);
19     startActivity(intent);
20 }
```

Listado 5.3: Estos métodos se encargan de cambiar de actividad y comunicar los datos de la hoja de cálculo.

- Primero se crea un objeto de la clase *Locale*, que será el encargado de guardar el nuevo idioma. Este se inicializa obteniendo el valor de la etiqueta del botón, como se observa en la línea 2 del Listado 5.4
- Luego, utilizando el objeto *Locale* que se ha creado, hay que actualizar la configuración para la app con él. Para ello se requiere tener la configuración en una variable e inicializarla *Configuration config = new Configuration()* de manera que se pueda modificar ahora el *Locale* que utiliza ésta, por el nuevo que se ha creado.
- No basta con tener una configuración con el idioma que se desea, ya que la configuración por defecto no ha cambiado, para esto hay que actualizarla para toda la app. Para lo cual se utiliza el método *updateConfiguration()* el cual se muestra en la línea 5 del Listado 5.4.
- Con esto ya se ha actualizado la configuración de idioma, y mostramos un mensaje al usuario, en el idioma seleccionado, de qué se ha cambiado.
- Finalmente, se invoca el método *recreate()* para actualizar el contenido de la ventana al idioma seleccionado.

```
1    public void changeLanguage(View view) {
2        Locale locale = new Locale(view.getTag().toString());
3        locale.setDefault(locale);
4        config.setLocale(locale);
5        this.getResources().updateConfiguration(config,this.getResources().getDisplayMetrics());
6        Toast toast = Toast.makeText(getApplicationContext(), getString(R.string.changelang), Toast
             .LENGTH_SHORT);
7        toast.show();
8        Intent intent = new Intent(this, MainActivity.class);
9        intent.putExtra("Datos", (Serializable) Datos);
10       startActivity(intent);
11   }
```

Listado 5.4: Método encargado de cambiar y actualizar el idioma de la app.

Capítulo 6

Configuración común de los mapas

En este capítulo se explicarán los métodos de la clase encargada de gestionar todo aquello que es común a todos los mapas:

- Configurar las *InfoWindows* para mostrar imágenes en ellas.
- Llevar control de un *Callback* que se encargue de actualizar las *InfoWindows* al cargar la imagen que tenga asociada.
- La creación del menú superior, desde el cual se podrán visitar URLs relacionadas con el edificio seleccionado, situar al usuario en un edificio de un campus y cambiar de tipo de mapa.
- Gestionar las acciones a realizar según qué opción del menú se seleccione.

6.1. InfoWindow personalizadas

Para crear una *InfoWindow* o ventana de información personalizada, lo primero que se necesita es crear un archivo XML que defina su contenido. Se puede ver el utilizado para `ULL-Maps` en el Listado 6.1

Una vez se tiene la base de la *InfoWindow* definida en el fichero XML, hay que indicarle a la app que se quiere utilizar una plantilla personalizada y no la que viene por defecto, para ello se utiliza el método que se muestra en el Listado 6.2 el cual se explicará a continuación.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:id="@+id/place_icon"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:focusable="false" />

    <TextView
        android:id="@+id/place_title"
        android:textColor="#000000"
        android:textStyle="bold"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/place_snippet"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/more_info"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/more_info"
        android:textColor="#000000"
        android:textStyle="italic" />

</LinearLayout>
```

Listado 6.1: Archivo XML que contiene la configuración de las ventanas de información.

Primero hay que crear una vista utilizando el código del fichero XML que se quiere que muestre la *InfoWindow*. En el caso de `ULL-Maps`, se utiliza el código que se presenta en el Listado 6.1. A continuación, se guardan los campos del fichero XML, los *TextView* en los que se mostrará el título, la descripción y uno informativo y el *ImageView* que será el encargado de mostrar una imagen que facilite la identificación del edificio.

Una vez se tiene todo lo mencionado anteriormente listo, se puede empezar a establecer cómo se desea que se rellene ese contenido. Para ello utilizando los marcadores, ya que las ventanas de información están asociadas a éstos, es decir, cada marcador tiene su ventana propia, se utiliza la información almacenada en éstos para establecer los textos:

- El título que tiene el marcador en el *TextView* título.
- El snippet en el *TextView* descripción, se debe tratar primero, ya que en el *snippet* del marcador se encuentra en todos los idiomas de la app y sólo se

```
1
2       mMap.setInfoWindowAdapter(new GoogleMap.InfoWindowAdapter() {
3           @Override
4           public View getInfoWindow(Marker marcador) {
5               return null;
6           }
7           //Definimos el contenido de nuestra infoWindows personalizada
8           @Override
9           public View getInfoContents(Marker marcador) {
10              // Obtenemos la vista del layout de nuestra infowindow personalizada
11              @SuppressLint("InflateParams") View v = getLayoutInflater().inflate(R.layout.
                    custom_infowindow, null);
12              //Apuntamos a las vistas de nuestro layout para poder trabajar con ellas
13              TextView title = (TextView) v.findViewById(R.id.place_title);
14              TextView description = (TextView) v.findViewById(R.id.place_snippet);
15              ImageView image = (ImageView) v.findViewById(R.id.place_icon);
16              //Les asignamos sus valores usando el marcador
17              title.setText(marcador.getTitle());
18              //Seleccionamos que texto usar segun el idioma
19              String languageText[] = marcador.getSnippet().split(":language:");
20              //Preparamos un String para quedarnos con una descripcion breve
21              String basicText[];
22      //Creamos un switch que depene de la lengua en la que este la app
23              switch (Locale.getDefault().getLanguage().toString()) {
24                  case "es":
25                      basicText = languageText[0].split(":basico:");
26                      if (basicText.length > 3)
27                          description.setText(basicText[1] + basicText[3] + markepos +
28                      getString(R.string.more_info));
29                      else
30                          description.setText(languageText[0].replace(":basico:", "") + markepos
                                + getString(R.string.more_info));
31                      break;
32                  ...
33      }
34
35              //Manejamos el refresco de la infoWindow y cargamos las imagenes si las hay
36              if (markerData.get(marcador).getImage().compareTo("") != 0) {
37                  if (markerData.get(marcador).getVisited()) {
38                      Picasso.with(context)
39                              .load(markerData.get(marcador).getImage())
40                              .resize(1000,500)
41                              .into(image);
42                  } else {
43                      markerData.get(marcador).setVisited(true);
44                      Picasso.with(context)
45                              .load(markerData.get(marcador).getImage())
46                              .resize(1000,500)
47                              .into(image, new InfoWindowRefresher(marcador));
48                  }
49              }
50              return v;
51
52          }
53      });
```

Listado 6.2: Método encargado de cambiar la ventana de información por defecto por una personalizada.

```
1  //Metodo que actualiza las infowindow al cargarse la imagen
2  public class InfoWindowRefresher implements Callback {
3         private Marker markerToRefresh;
4
5         InfoWindowRefresher(Marker markerToRefresh) {
6             this.markerToRefresh = markerToRefresh;
7         }
8
9         @Override
10        public void onSuccess() {
11            markerToRefresh.showInfoWindow();
12        }
13
14        @Override
15        public void onError() {}
16    }
```

Listado 6.3: Método que maneja el callback de la carga de imágenes.

tiene que mostrar esta información en uno de ellos y además sólo una parte de ellos. Se puede observar cómo se desglosa el texto, en el caso del español, en las líneas 20-32 del Listado 6.2.

- El texto informativo es el encargado de indicar que la ventana de información se puede seleccionar para obtener más información. El texto de ésta está definido por la aplicación para cada idioma.

Y además de los textos también se define la imagen que se tiene que mostrar. Para ello lo primero que se tiene que hacer es comprobar que para ese marcador se dispone de una URL con una imagen. Si así es, utilizando la librería *Picasso* [**?**], la cual gestiona la descarga y almacenamiento volátil de imágenes, se procederá a la carga de la imagen, pero se debe tener en cuenta que hay dos casos posibles dependiendo de si la imagen ha sido descargada o no.

En el primer caso al abrir la *InfoWindow* no se podrá ver la imagen y aunque ésta se descargue después no se mostrará al usuario hasta que vuelva a abrir la *InfoWindow*. Para evitar este problema se le indica a la librería que se ejecute un *callback* que se especifica en el código, en `ULL-Maps` es el que se puede observar en el Listado 6.3. En el segundo caso sólo es necesario indicarle de donde se quiere descargar la imagen y donde se quiere colocar.

Y con esto ya está configurada la ventana de información personalizada, que dependerá de la información de su marcador asociado.

6.2. Diseño del menú superior

A continuación se explica cómo crear el menú superior del que dispone cada ventana de la app.

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:id="@+id/map"
        android:name="com.google.android.gms.maps.SupportMapFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context="com.example.tareq.tfg_googlemaps_ull.MapsActivity" />

</RelativeLayout>
```

Listado 6.4: XML encargado del diseño de la ventana mapa.

Primero es necesario modificar el layout que se crea al seleccionar en Android Studio [4] ya que éste únicamente permite ver el mapa, y se desea mostrar además un menú. Para poder añadir el menú sólo es necesario crear un *RelativeLayout* y dentro de éste incluir un *FragmentLayout* que contenga el mapa, quedando finalmente un fichero XML como el que se muestra en el Listado 6.4

Ahora que ya se tiene la base desde la que trabajar y puesto que se quiere usar el menú para los distintos tipos de mapa, se tiene que crear un fichero XML que represente al menú que se desea utilizar. El menú deberá tener un item por cada opción que se quiera mostrar. para `ULL-Maps` se utilizan tres, que se pueden ver en el Listado 6.5.

Con todo esto hecho sólo queda incluir el menú en los layout de los mapas. Esto se realizará programáticamente al igual que la ampliación de las subopciones del menú, ya que dependen de la información que se encuentra en la hoja de cálculo que utiliza la aplicación. Ahora se explicará qué se quiere mostrar según cual de los tres item del menú se seleccionen:

- Menu_option, que en la app se denomina Enlaces. Al ser seleccionada según se abra la actividad indicará al usuario que seleccione un edificio. Si el usuario ya ha seleccionado un edificio mostrará distintas URL asociadas a ese edificio.

- Menu_option2, que en la app se denomina Campus. Al ser seleccionada mostrará los distintos campus de la ULL, y al elegir uno de ellos se podrá seleccionar un edificio perteneciente a ese campus, localizando al usuario sobre el edificio seleccionado y resaltando el mismo mediante un círculo colocado sobre él.

- Menu_option3, que en la app se denomina Mapa, mostrará al usuario tres opciones que varían según en qué tipo de mapa se encuentre. Las opciones

que muestra son:

- Inicio: devuelve al usuario a la página inicial de la aplicación.
- Marcadores: cambia el mapa actual por uno con marcadores.
- Iconos: cambia el mapa actual por uno con iconos.
- Polígonos: cambia el mapa actual por uno con polígonos.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/topMenu"
        app:showAsAction="ifRoom"
        android:title="@string/menu_option">
        <menu>
            <group android:menuCategory="container"/>
        </menu>
    </item>
    <item
        android:id="@+id/topMenu2"
        app:showAsAction="ifRoom"
        android:title="@string/menu_option2">
        <menu>
            <group android:menuCategory="container"/>
        </menu>
    </item>
    <item
        android:id="@+id/topMenu3"
        app:showAsAction="ifRoom"
        android:title="@string/menu_option3">
        <menu>
            <group android:menuCategory="container"/>
        </menu>
    </item>
</menu>
```

Listado 6.5: XML encargado del diseño del menú superior que tienen las ventanas de mapas de la app.

Como se puede observar todo esto no se encuentra en el fichero XML, sino que se implementa en el código de la aplicación. Además no se podría realizar de forma eficiente ya que aunque se incluyeran todos los edificios de todos los campus, si en el futuro se construyen nuevos edificios se tendría que modificar el fichero XML. En cambio si se hace mediante código utilizando los datos de una hoja de cálculo, sólo será necesario añadir la información a ésta y el menú se actualizará para la aplicación sin necesidad de modificar ésta.

A continuación se procederá a explicar cómo se configura el menú, y cómo se utiliza la información de la hoja de cálculo para alimentar sus distintas subopciones. Lo primero a tener en cuenta es la creación de un objeto *Menu* que será sobre el que se trabajará. Una vez creado se procede a indicarle que utilice

como base el fichero XML que se creó anteriormente. Para ello se obtendrá el *MenuInflater*, que es el objeto encargado de instanciar ficheros de menú XML en objetos de tipo menú, y se le indica que utilice el código que se muestra en el Listado 6.5. Una vez hecho esto se puede comenzar a configurar el contenido de éste, pudiendose ver el código necesario para realizar esta configuración del menú, para `ULL-Maps`, en el Listado 6.6:

1. Se tiene que configurar un texto que se muestre al seleccionar la opción *Enlaces*. Esto se consigue añadiendo una subopción y asignándole un texto, con ello se logra que el usuario sepa que el botón funciona y sepa cómo proceder. En el caso de `ULL-Maps` se le indica que debe seleccionar un edificio (Línea 9).

2. Se agrega a la opción de *Campus* una subopción por cada campus de la Universidad (Líneas 11-15), teniendo en cuenta que habrá que asignarle a cada una un identificador para poder añadir los edificios correspondientes a su campus. Para ello posteriormente hay que pasarle al método *addSubmenu* los parámetros:

   - `GroupID`: un identificador de grupo. En el caso de `ULL-Maps` no se usan grupos para el menú por lo que se le asigna el valor *Menu.NONE*.
   - `ItemID`: un identificador para el subMenú agregado de manera que se pueda acceder al mismo utilizando su ID. Aprovechando esta funcionalidad para luego poder agregar todos los edificios a su campus correctamente.
   - `Orden`: por si se desea que las opciones se dispongan en un orden específico. En el caso de nuestra aplicación no se le asigna ningún valor.
   - `Nombre`: cómo se quiere que aparezca la opción en el menú.

3. Al igual que en el segundo paso se definen los campus, hay que definir las subopciones para cambiar de mapa o volver a la pantalla inicial de la aplicación, por lo que en total son cuatro opciones (Líneas 17-20).

4. El último paso que se puede realizar de manera común a los mapas, es agregar los edificios a sus campus correspondientes, esto se realiza utilizando un *switch* para saber en qué campus se localiza el edificio (Líneas 29-45). Para que esto funcione se debe indicar en la hoja de cálculo para cada edificio a qué campus pertenece, de manera que se pueda identificar.

En el cuarto paso además se aprovecha y según se crean las distintas subopciones también se guarda en un *HashMap<Edificio, Posición>* las distintas localizaciones de cada edificio (Línea 27), para que posteriormente al seleccionar el edificio como opción se pueda situar al usuario sobre el mismo.

```
1   public boolean onCreateOptionsMenu(Menu menu) {
2       menuID = menu;
3       String campus;
4       String edificio;
5       LatLng posicion;
6       MenuInflater inflater = getMenuInflater();
7       inflater.inflate(R.menu.map_menu, menu);
8       //Primer paso
9       menu.findItem(R.id.topMenu).getSubMenu().add(Menu.NONE, 0, Menu.NONE, getString(R.string.
            urloption));
10      //Segundo paso
11      menu.findItem(R.id.topMenu2).getSubMenu().addSubMenu(Menu.NONE, 0, Menu.NONE, "Guajara");
12      menu.findItem(R.id.topMenu2).getSubMenu().addSubMenu(Menu.NONE, 1, Menu.NONE, "Anchieta");
13      menu.findItem(R.id.topMenu2).getSubMenu().addSubMenu(Menu.NONE, 2, Menu.NONE, "Santa Cruz")
            ;
14      menu.findItem(R.id.topMenu2).getSubMenu().addSubMenu(Menu.NONE, 3, Menu.NONE, "Central");
15      menu.findItem(R.id.topMenu2).getSubMenu().addSubMenu(Menu.NONE, 4, Menu.NONE, "Externos");
16      //Tercer paso
17      menu.findItem(R.id.topMenu3).getSubMenu().add(Menu.NONE, 5, Menu.NONE, "Inicio");
18      menu.findItem(R.id.topMenu3).getSubMenu().add(Menu.NONE, 6, Menu.NONE, "Poligonos");
19      menu.findItem(R.id.topMenu3).getSubMenu().add(Menu.NONE, 7, Menu.NONE, "Iconos");
20      menu.findItem(R.id.topMenu3).getSubMenu().add(Menu.NONE, 8, Menu.NONE, "Marcadores");
21      //Cuarto paso
22      for (List row : Datos) {
23          campus = row.get(GlobalVariables.getCAMPUS()).toString();
24          edificio = row.get(GlobalVariables.getTITLE()).toString();
25          posicion = new LatLng(Double.parseDouble(row.get(GlobalVariables.getLAT_POS()).toString
                ()),
26                  Double.parseDouble(row.get(GlobalVariables.getLNG_POS()).toString()));
27          locateBuild.put(edificio, posicion);
28
29          switch(campus) {
30              case "Guajara":
31                  menu.findItem(R.id.topMenu2).getSubMenu().getItem(0).getSubMenu().add(edificio);
32                  break;
33              case "Anchieta":
34                  menu.findItem(R.id.topMenu2).getSubMenu().getItem(1).getSubMenu().add(edificio);
35                  break;
36              case "Santa Cruz":
37                  menu.findItem(R.id.topMenu2).getSubMenu().getItem(2).getSubMenu().add(edificio);
38                  break;
39              case "Central":
40                  menu.findItem(R.id.topMenu2).getSubMenu().getItem(3).getSubMenu().add(edificio);
41                  break;
42              case "Externos":
43                  menu.findItem(R.id.topMenu2).getSubMenu().getItem(4).getSubMenu().add(edificio);
44                  break;
45          }
46      }
47      return super.onCreateOptionsMenu(menu);
48  }
```

Listado 6.6: Código en Java que se encarga de configurar el menú superior de la aplicación `ULL-Maps`.

6.3. Control de las opciones del menú superior

Ahora que ya se dispone del menú, se tiene que indicar qué operaciones se deben realizar al seleccionar cada una de las opciones. En el caso de la aplicación objeto de este TFG se tendrán los siguientes casos:

- Se busca cambiar de mapa, por lo que ha seleccionado una subopción de *Mapa*, las cuales pueden ser:
  - Inicio.
  - Polígonos. Esta opción no se muestra si el usuario se encuentra en el mapa de polígonos.
  - Iconos. Esta opción no se muestra si el usuario se encuentra en el mapa de iconos.
  - Marcadores. Esta opción no se muestra si el usuario se encuentra en el mapa de marcadores.

  Teniendo esto en cuenta se utiliza una sentencia *switch* y se obtiene el identificador de la opción seleccionada utilizando el método *getItemId()* del objeto. Una vez se tiene el ID del objeto, se procede a compararlo para saber si la opción seleccionada se refiere a un cambio de mapa (valores 5, 6 ,7 y 8), si es así, se crea un nuevo Intent y se cambia a la pantalla correspondiente. Véase líneas 11-31 en el Listado 6.7.

- Se quiere localizar un edificio, por lo que se encuentra en una subopción de *Campus*, esta sería la opción por defecto del *switch* utilizado en el caso anterior, en este caso se tiene que comprobar si para el *HashMap<Edificio, Posición>* hay un valor para el edificio seleccionado:
  - Si existe, se mueve la cámara y se centra sobre el mismo. Además, para poder identificarlo correctamente, se dibuja un círculo sobre el edificio, de manera que el usuario tenga claro cual es el que ha seleccionado.
  - Si no existe, es el tercer caso posible de opción seleccionada.

- Último caso posible, el usuario ha seleccionado una URL. Lo primero que se debe hacer es comprobar que existe una URL para esa opción, ya que al usuario no se le muestra la dirección URL sino un nombre representativo de ésta, de modo que le resulte más sencillo saber a dónde se dirige. Una vez se comprueba que hay una URL disponible se procede a crear un nuevo Intent de navegador o BrowserIntent con la URL que se ha obtenido, mostrándole al usuario en el navegador de su dispositivo móvil la página que seleccionó en el menú.

Se puede ver el manejo de las distintas opciones en el código del Listado 6.7, donde se puede observar además que cuando se dibuja el círculo en el mapa se guarda en una variable su valor, de modo que al seleccionar otro edificio se pueda eliminar el círculo anterior.

6.4. MarkersData

Los marcadores de Google Maps [7] por defecto sólo pueden guardar la siguiente información:

- Título. En `ULL-Maps` será lo primero que se muestra en la *InfoWindow* asociada al marcador y estará resaltado en negrita.
- Snippet: contendrá un texto breve que se situará debajo del título en la *InfoWindow*.
- Posición: para poder crear un marcador es necesario dar las coordenadas en las que se desea posicionar el mismo.

Para la aplicación `ULL-Maps` se pretende mostrar además una imagen, y para cada marcador guardar distintas URL asociadas al mismo, por lo que no es suficiente con esta información. Para solventar este problema se utiliza una clase *MarkersData* que será la encargada de almacenar esta información. Para poder asociar un objeto de esta clase a cada marcador se utiliza un *HashMap<Marker, MarkerData>* de forma que se pueda acceder al objeto de la clase para cada marcador. El código de la clase se presenta en el Listado 6.8.

```
1  class MarkersData {
2      private String urls;
3      private String image;
4      private Boolean visited;
5
6      MarkersData () {
7          urls = "Sin URL";
8          image = "";
9          visited = false;
10     }
11
12     String getImage() { return image; }
13     Boolean getVisited() { return visited; }
14     String getUrls() { return urls; }
15     void setImage(String image) { this.image = image; }
16     void setUrls(String urls) { this.urls = urls; }
17     void setVisited(Boolean visited) { this.visited = visited;}
18 }
```

Listado 6.8: Clase encargada de gestionar la información adicional de cada marcador.

```java
public boolean onOptionsItemSelected(MenuItem item) {
    int opcion = item.getItemId();
    String edificio = item.getTitle().toString();
    String url;
    LatLng posicion;
    CameraUpdate camUpd1;
    Intent intent;

    url = option_Url.get(edificio);
    posicion = getLocateBuild().get(edificio);
    switch (opcion) {
        case 6: //Poligonos
            intent = new Intent(this, MapsActivity_Poligons.class);
            intent.putExtra("Datos", (Serializable) Datos);
            startActivity(intent);
            break;
        case 7: //Iconos
            intent = new Intent(this, MapsActivity_Icons.class);
            intent.putExtra("Datos", (Serializable) Datos);
            startActivity(intent);
            break;
        case 8: //Marcadores
            intent = new Intent(this, MapsActivity.class);
            intent.putExtra("Datos", (Serializable) Datos);
            startActivity(intent);
            break;
        case 5: //Inicio
            intent = new Intent(this, MainActivity.class);
            intent.putExtra("Datos", (Serializable) Datos);
            startActivity(intent);
            break;
        default: //Url o cambio de posicion
            if (url != null) {
                url = url.replace(" ", "");
                if (url.startsWith("https:")) {
                    Intent browserIntent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
                    startActivity(browserIntent);
                }
            }
            else if (getLocateBuild().get(edificio) != null) {
                camUpd1 = CameraUpdateFactory
                        .newLatLngZoom(posicion, GlobalVariables.getBUILD_ZOOM());
                mMap.moveCamera(camUpd1);
                if (circle != null)
                    circle.remove();
                circleOptions.center(posicion);
                circle = mMap.addCircle(circleOptions);
            }
    }

    return true;
}
```

Listado 6.7: Código en Java que maneja las acciones a realizar según que opción del menú superior se seleccione

Capítulo 7

Ventana *FullInfo* y los mapas

En este capítulo se explicarán los distintos tipos de mapeado y la ventana que se utiliza para mostrar toda la información disponible de un edificio.

7.1. Ventana *FullInfo*

Como las *InfoWindow* están limitadas en cuanto a la cantidad de información que pueden mostrar al no disponer de una *scrollbar* que permita desplazarse, es necesario buscar una alternativa que permita al usuario de la aplicación, obtener toda la información disponible. Teniendo esto en cuenta y puesto que se quiere incluir, al menos, los siguientes datos sobre cada edificio: los grados que se imparten, los servicios que se ofertan, el horario de apertura y los datos de contacto.

Se crea una nueva actividad, en la que se muestra toda esta información, permitiendo además en un futuro añadir más datos si fuera preciso. Para ello lo primero será crear un layout con una vista del tipo *ScrollView*, ya que no se quiere perder información y al tener la ventana la capacidad de desplazarse dentro de ella (*scrolling*) se puede visualizar toda la información que contenga. Además dentro de esta vista se tendrá otra, en este caso del tipo *RelativeLayout* de manera que se puedan posicionar los distintos elementos, como por ejemplo las *TextView*, donde se desee.

Se puede ver el código de este fichero XML en el Listado 7.1. En la línea 19 del mismo se puede observar la opción *autolink* con el valor *all*. Con esto se consigue que se detecte cualquier número o e-mail que exista en el texto asociado al *TextView*. Estos textos (número, e-mail) se muestran resaltados y es posible seleccionarlos para, en el caso de un número de teléfono llamar al mismo o, si es un e-mail, enviar un correo a ese destinatario.

Con lo dicho anteriormente se abarca la parte contenido, pero falta cargar en

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <ImageView
            android:id="@+id/place_icon"
            ... />

        <TextView
            android:id="@+id/place_title"
            ... />

        <TextView
            android:id="@+id/place_snippet"
            ...
            android:autoLink="all"
            android:text="Descripcion" />
    </RelativeLayout>
</ScrollView>
```

Listado 7.1: Este es el fichero XML encargado de mostrar toda la información de un edificio.

su campo correspondiente la información deseada y crear un menú que permite al usuario tanto ver y acceder a las distintas URL asociadas al edificio que se está examinando, como una opción para volver al mapa en el que se encontraba antes de cambiar de ventana.

El menú para esta ventana sólo comparte con el de los mapas el campo de los enlaces y además tendrá otra opción para poder volver a la ventana anterior de la aplicación. Al ser distinto este menú del que se mostró en el Listado 6.5, es necesario crear un nuevo fichero XML, que se puede ver en el Listado 7.2.

Con esto ya realizado queda explicar el código encargado de rellenar la información, ya que esta dependerá del marcador que se haya seleccionado. Lo primero que se necesita es pasarle a la actividad la información que requiere. Para ello se sobreescribe el método *onInfoWindowClick(Marker marker)*, de manera que se pueda gestionar la acción a realizar al seleccionar una ventana de información. Para `ULL-Maps` se busca mostrar la siguiente información sobre el edificio que represente:

- Nombre.
- Descripción, al menos en español.
- Dirección URL que contenga una imagen del edificio.
- Enlaces asociados.

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/enlaces"
        app:showAsAction="ifRoom"
        android:title="@string/menu_option">
        <menu>
            <group android:menuCategory="container"/>
        </menu>
    </item>
    <item
        android:id="@+id/back"
        app:showAsAction="ifRoom"
        android:title="@string/back">
        <menu>
            <group android:menuCategory="container"/>
        </menu>
    </item>
</menu>
```

Listado 7.2: Fichero XML del menú de la actividad que muestra la información completa de un edificio.

- Posición en latitud y longitud.

Toda esta información se obtiene del marcador, pero puesto que se requieren en forma de objetos *String* se pasan a la actividad como tal para luego procesarlas en ella. Se puede observar cómo se implementa esta labor en el Listado 7.3.

Finalmente queda explicar qué acciones se realizan en la ventana encargada de mostrar toda la información. Lo primero sería rellenar el contenido de la vista con la información que se le pasa a la actividad al seleccionar una ventana de información en un mapa. Para ello se edita el método *onCreate()* y se indican en él las siguientes acciones a realizar:

```java
mMap.setOnInfoWindowClickListener(new GoogleMap.OnInfoWindowClickListener() {
          @Override
          public void onInfoWindowClick(Marker marker) {
              Intent intent = new Intent(context, FullInfoActivity.class);
              intent
                      .putExtra("Imagen", getMarkerData().get(marker).getImage().replace(" ", ""))
                      .putExtra("Titulo", marker.getTitle())
                      .putExtra("Snippet", marker.getSnippet())
                      .putExtra("URLs", getMarkerData().get(marker).getUrls())
                      .putExtra("POS", marker.getPosition());
              startActivity(intent);
          }
      });
```

Listado 7.3: Creación de un listener que recoja la selección en *InfoWindows* y la acción a realizar al producirse el evento.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_full_info);
    TextView titulo = (TextView) this.findViewById(R.id.place_title);
    TextView snippet = (TextView) this.findViewById(R.id.place_snippet);
    ImageView image = (ImageView) this.findViewById(R.id.place_icon);
    String posicion = getIntent().getExtras().get("POS").toString();
    String languageText[] = getIntent().getExtras().get("Snippet")
            .toString()
            .replace(":basico:", "")
            .split(":language:");
    titulo.setText(getIntent().getExtras().get("Titulo").toString());

    switch (Locale.getDefault().getLanguage().toString()) {
        case "es":
            snippet.setText(languageText[0] + posicion);
            break;
        case "en":
            if (languageText.length > 1)
                snippet.setText(languageText[1] + posicion);
            else
                snippet.setText(languageText[0] + posicion);
            break;
            ...
    }

    if (getIntent().getExtras().get("Imagen").toString().startsWith("http")) {
        Picasso.with(this)
                .load(getIntent().getExtras().get("Imagen").toString())
                .resize(1000, 500)
                .into(image);
    }

}
```

Listado 7.4: Inicialización de la ventana *FullInfo* de la app.

- Registrar en variables los distintos campos del fichero XML del Listado 7.1 para poder editar su contenido.
- Guardar en *Strings* los datos que se le pasan a la actividad y formatearlos según el idioma elegido por el usuario en la ventana principal de la app.
- Actualizar el valor de los campos de texto una vez ya se ha recogido y tratado la información.
- Cargar la imagen asociada al edificio que se describe utilizando la librería *Picasso*, la URL donde encontrarla y la variable que apunta al *ImageView*.

Se puede observar cómo se realiza este proceso en el Listado 7.4.

En este punto ya se tiene todo el contenido interno y queda por configurar el menú de esta ventana. Para ello hay que realizar dos puntos al igual que se hizo en su momento para los menú de los mapas:

```
public boolean onCreateOptionsMenu(Menu menu) {
    String[] urls = getIntent().getExtras().get("URLs").toString().split(";");
    String[] nameAsign;

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.full_info_menu, menu);

    for (String url: urls ) {
        nameAsign = url.split("-");
        if(nameAsign.length > 1)
            option_Url.put(nameAsign[0], nameAsign[1]);
        else
            option_Url.put(nameAsign[0], nameAsign[0]);
        menu.findItem(R.id.enlaces).getSubMenu().add(Menu.NONE, 1, Menu.NONE, nameAsign[0]);
    }
    return super.onCreateOptionsMenu(menu);
}
```

Listado 7.5: Manejo de los eventos para las opciones del menú de la ventana *FullInfo*.

1. Configurar el menú y sus subopciones, lo cual se consigue obteniendo el *MenuInflater* e indicándole que tome como referencia el XML que se quiere. En este caso es el que se muestra en el Listado 7.2. Partiendo del *String* que contiene las URL asociadas, se crean las distintas subopciones en tiempo de ejecución, actualizándose éstas al seleccionar un edificio. Se puede observar el método encargado de esta funcionalidad en el Listado 7.6.

2. Manejar los eventos según qué opción se ha seleccionado, lo cual se consigue comprobando primero si la opción es la de volver, en cuyo caso se cierra la ventana con el método *finish()* y en otro caso, se comprueba que es una URL y si lo es se abre el navegador del dispositivo apuntando a esa dirección. Se puede estudiar el código que gestiona estos eventos en el Listado 7.5.

Con esto queda explicada la ventana encargada de mostrar la información detallada de cada edificio.

7.2. Mapas

Ahora se procederá a explicar los casos particulares de cada mapa, teniendo en cuenta que todos parten de lo expuesto en las secciones anteriores, y se comentará en qué se diferencian.

```
public boolean onOptionsItemSelected(MenuItem item) {
    String opcion = item.getTitle().toString();
    String url = option_Url.get(opcion);
    if (opcion.compareTo("back") == 0)
        finish();
    else {
        if (url != null)
            url = url.replace(" ", "");
        else
            url = "null";
        // Handle item selection
        if (url.startsWith("https:") ) {
            Intent browserIntent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
            startActivity(browserIntent);
        }
    }
    return true;
}
```

Listado 7.6: Creación de las subopciones del menú para la ventana *FullInfo* de la app.

7.2.1. Mapa con marcadores

Los mapas con marcadores son los más sencillos, puesto que sólo hay que introducir los datos necesarios de cada marcador y posteriormente representarlos en el mapa.

Lo primero que se ha de hacer al crear el mapa es sobreescribir el método *onCreate()* y se realizarán los siguientes pasos dentro de éste:

1. Invocar al método *super.onCreate()* ya que se sobreescribirá y lo que se pretende es añadir nuevas funcionalidades sin eliminar las básicas.

2. Indicarle que se va a utilizar un layout propio. Esto se consigue con el método *setContentView()*.

3. Se obtiene el *SupportMapFragment* y se ejecuta sobre éste un *getMapAsync(this)* para saber cuando el mapa está listo para ser usado.

4. Como se ha indicado anteriormente, los datos de la aplicación se obtienen de una hoja de cálculo en la primera pantalla de la aplicación y de esta se envía a las demás como un objeto *Serializable*. Ahora se tiene que registrar este objeto en el mapa. Para obtener este objeto se utiliza el siguiente método *getIntent().getExtras().getSerializable(ID_del_objeto)*, sólo quedaría asignarle ese valor a una variable para poder utilizarla.

El método encargado de realizar estos pasos se puede estudiar en el Listado 7.7.

```
1   protected void onCreate(Bundle savedInstanceState) {
2       super.onCreate(savedInstanceState);
3   //Seleccionamos el layout que deseamos utilizar para esta ventana.
4       setContentView(R.layout.activity_maps);
5       //Obtenemos el SupportMapFragment y somos notificados cuando el mapa esta listo para ser
            utilizado.
6       SupportMapFragment mapFragment = (SupportMapFragment) getSupportFragmentManager()
7               .findFragmentById(R.id.map);
8       mapFragment.getMapAsync(this);
9       //Guardamos el valor del SpreadSheet para poder utilizar la informacion que contiene.
10      setDatos((List<List<Object>>) getIntent().getExtras().getSerializable("Datos"));
11      this.Datos = getDatos();
12  //Tipo 0 = Poligonos, tipo 1 = iconos, tipo 2 = marcadores.
13      setTipo(2);
14  }
```

Listado 7.7: Código en JAVA que se ejecuta a la hora de crear una actividad de la aplicación que tenga un mapa.

A continuación se tienen que definir distintas acciones que se deben realizar una vez el mapa esté listo, o lo que es lo mismo, sobreescribir el método *onMapReady()*. Lo primero será cargar los distintos marcadores del mapa en cuestión, para lo cual en `ULL-Maps` se utilizan dos métodos:

- *LoadMarkers()* (Véase el Listado 7.9): encargado de recorrer cada fila de información e invocar al segundo método para cada una de ellas.
- *CreateMarker()* (Véase el Listado 7.10): es el encargado de crear cada marcador, con sus datos de posición, título y descripción, y asignarle, si las tiene, una imagen del edificio que representa y las distintas URL que están asociadas al marcador en cuestión.

Además de representar los marcadores en el mapa, también se crea un *setOnMarkerClickListener()* para los marcadores, de manera que al seleccionar uno de ellos se actualicen los enlaces relacionados al edificio que representa ese marcador en el menú superior. Para ello se obtiene la cadena que contiene todos los enlaces con su nombre identificativo, se realiza un *split()* para separarlos y poder guardarlos en un *HashMap<Nombre, URL>*, y finalmente se actualiza el apartado de enlaces del menú superior. Se puede estudiar el código de este *Listener* en el Listado 7.8

Este tipo de mapeado es el más simple ya que no hay que hacer nada más que guardar la información necesaria de modo que luego se pueda acceder a ella.

7.2.2. Mapa con iconos

Los mapas con iconos están basados en los mapas con marcadores, con la diferencia de que son representados con un icono que representa el tipo de edificio

```
mMap.setOnMarkerClickListener(new GoogleMap.OnMarkerClickListener()
{

    @Override
    public boolean onMarkerClick(Marker marcador) {
        String[] urls = getMarkerData().get(marcador).getUrls().split(";");
        String[] nameAsign;
        getMenuID().findItem(R.id.topMenu).getSubMenu().clear();
        for (String url: urls ) {
            nameAsign = url.split("-");
            if(nameAsign.length > 1)
                getOption_Url().put(nameAsign[0], nameAsign[1]);
            else
                getOption_Url().put(nameAsign[0], nameAsign[0]);
            getMenuID().findItem(R.id.topMenu).getSubMenu().add(Menu.NONE, 1, Menu.NONE, nameAsign
                 [0]);
        }
        return false;
    }

});
```

Listado 7.8: Código encargado de actualizar el menú al seleccionar un marcador.

```
    public void loadMarkers() {
        //Para cada fila del SpreadSheet
        int fila = 0;
        for (List row : Datos) {
            createMarker(row.get(gb.getLAT_POS()).toString(), row.get(gb.getLNG_POS()).toString(),
                 fila);
            fila++;
        }
    }
```

Listado 7.9: Método encargado de cargar los marcadores.

```
1  public void createMarker(String lat, String lng, int fila) {
2         //Creamos el objeto que contiene la informacion que necesitamos de cada marcador.
3         MarkersData mData = new MarkersData();
4         //Convertimos las String en Double para poder crear un objeto LatLng.
5         LatLng posicion = new LatLng( Double.parseDouble(lat), Double.parseDouble(lng));
6         MarkerOptions markerOptions = new MarkerOptions()
7                 .position(posicion)
8                 .title(Datos.get(fila).get(gb.getTITLE()).toString())
9                 .flat(true)
10                .snippet(Datos.get(fila).get(gb.getDESC()).toString());
11
12        Marker marcador = getmMap().addMarker(markerOptions);
13    //Si existe una imagen asociada al marcador la guardamos.
14        if(Datos.get(fila).size() > gb.getIMAGE() && Datos.get(fila).get(gb.getIMAGE()) != null)
15            mData.setImage(Datos.get(fila).get(gb.getIMAGE()).toString());
16    //Si existen URL asociadas al marcador las guardamos.
17        if(Datos.get(fila).size() > gb.getURL() && Datos.get(fila).get(gb.getURL()) != null)
18            mData.setUrls(Datos.get(fila).get(gb.getURL()).toString());
19
20    //Guardamos en un HashMap el objeto MarkersData asociado al marcador.
21        getMarkerData().put(marcador, mData);
22    }
```

Listado 7.10: Método encargado de crear cada marcador, y de almacenar los valores que necesita.

al que se refieren.

Al igual que en el mapa de marcadores, la primera tarea es modificar el método *onCreate()*, cambiando el *layout* a utilizar y el tipo de mapa. El resto de tareas son las mismas que en el caso de mapas con marcadores, se puede estudiar el código en el Listado 7.7.

Una vez completadas estas tareas, se procede a definir qué acciones deben realizarse una vez el mapa esta creado, por lo que se debe sobreescribir el método *onMapReady()*, al igual que se hizo en el mapa con marcadores, para cargar los distintos marcadores con sus iconos correspondientes. Para ello se utilizan dos métodos distintos:

- *LoadIcons()* (Véase el Listado 7.11): es el encargado de recorrer cada fila de información e invocar al segundo método para cada una de ellas.
- *CreateM_Icon()* (Véase el Listado 7.12): además de crear los marcadores con toda la información que requiere, como realiza el método *createMarker()* (Véase el Listado 7.10), se encarga de asignar a cada marcador un icono que lo represente, que estará definido por el tipo de edificio asociado a él.

Al igual que en el mapa con marcadores, también es necesario crear un *setOnMarkerClickListener()* (Véase el Listado 7.8) de manera que se actualice el menú superior. El contenido de éste es el mismo que en el caso del mapa con marcadores.

```
1  public void loadIcons() {
2         //Para cada fila del SpreadSheet
3         int fila = 0;
4         for (List row : getDatos()) {
5             createM_Icon(row.get(gb.getLAT_POS()).toString(), row.get(gb.getLNG_POS()).toString(),
                  fila);
6             fila++;
7         }
8     }
```

Listado 7.11: Método encargado de cargar los marcadores con iconos.

```
1  public void createM_Icon(String lat, String lng, int fila) {
2     //Creamos el objeto que contiene la informacion que necesitamos de cada marcador
3     MarkersData mData = new MarkersData();
4     //Convertimos las String en Double para poder crear un objeto LatLng
5     LatLng posicion = new LatLng( Double.parseDouble(lat), Double.parseDouble(lng));
6     String snippet = Datos.get(fila).get(gb.getDESC()).toString();
7     MarkerOptions markerOptions = new MarkerOptions()
8             .position(posicion)
9             .title(Datos.get(fila).get(gb.getTITLE()).toString())
10            .flat(true);
11
12    if (Datos.get(fila).get(gb.getICON()) != null) {
13        switch (Datos.get(fila).get(gb.getICON()).toString()) {
14            case "biblio":
15                markerOptions
16                        .snippet(snippet)
17                        .icon(BitmapDescriptorFactory.fromResource(R.drawable.biblioicon));
18                break;
19            case "centro":
20                markerOptions
21                        .snippet(snippet)
22                        .icon(BitmapDescriptorFactory.fromResource(R.drawable.centroicon));
23                break;
24            ...
25            default:
26                markerOptions
27                        .snippet(snippet)
28                        .icon(BitmapDescriptorFactory.fromResource(R.drawable.defaulticon));
29                break;
30        }
31    }
32    Marker marcador = getmMap().addMarker(markerOptions);
33    if(Datos.get(fila).size() > gb.getIMAGE() && Datos.get(fila).get(gb.getIMAGE()) != null)
34        mData.setImage(Datos.get(fila).get(gb.getIMAGE()).toString());
35    if(Datos.get(fila).size() > gb.getURL() && Datos.get(fila).get(gb.getURL()) != null)
36        mData.setUrls(Datos.get(fila).get(gb.getURL()).toString());
37
38    getMarkerData().put(marcador, mData);
39 }
```

Listado 7.12: Método encargado de crear cada marcador, y de guardar los valores que necesita.

Este tipo de mapeado es algo más complejo que el de marcadores, pero facilita al usuario la búsqueda de edificios al estar clasificados por iconos.

7.2.3. Mapa con polígonos

Los mapas con polígonos son los más sofisticados, ya que estos no son del tipo *Marker*. Son otro tipo de objeto denominado *Polygon* que ofrece la API de Google Maps [7], que permite representar en el mapa, dados unos vértices, un polígono.

La primera tarea consiste en definir el contenido del método *onCreate()* (Véase el Listado 7.7) que, al igual que sucede con el mapa con iconos, sólo cambia en el *Layout* a utilizar y en el tipo de mapa del que se trata.

Con el mapa creado, al igual que en los otros tipos de mapa, es necesario representar los polígonos en el mapa, de manera que sean visibles. La creación de estos polígonos se realiza una vez el mapa esté listo, es decir, en el método *onMapReady()* en el cual se realizan dos acciones:

- La carga de los polígonos. Ésta se realiza utilizando dos métodos:
  - *LoadPolygons()* (Véase el Listado 7.13): al contrario que en el mapa con marcadores o iconos, en este caso se requiere de varios vértices para poder crearlo. Éstos se obtienen de la hoja de cálculo que utiliza la aplicación [10]. Como no se puede saber a priori el número de vértices que se necesitan para cada edificio, éstos se obtienen desde una columna hasta que no exista más información en la fila y se guarda la información de cada celda en un *ArrayList<LatLng>* (Línea 24 en el Listado 7.13).
  - *createPolygon()* (Véase el Listado 7.14): este método necesita el *ArrayList<LatLng>* que contiene los vértices del polígono, un punto que represente el centro del edificio y la fila de la hoja de cálculo que contiene la información del edificio que representa. Partiendo de estos datos que obtiene del método *LoadPolygon()*, se definen las *PolygonOptions* con las que se representa un polígono en el mapa. Primero se introducen en éstas todos los vértices, se indica que es seleccionable (Línea 12 en el Listado 7.14) y se puede cambiar tanto el color del interior como de la línea exterior opcionalmente. Con esto se puede crear el objeto *Polygon* e introducirlo en el mapa utilizando el método de la API *addPolygon()* sobre el mapa. Además, es necesario crear un marcador, de manera que al seleccionar un polígono se disponga de una ventana de información ya que la API sólo permite a los *Markers* utilizarlas. Este proceso es el mismo que se realiza para el mapa con marcadores, con la diferencia de que no se representan en el mapa, se mantienen ocultos y se guardan en un *HashMap<Polygon, Marker>* de manera que se tenga acceso al marcador asociado a cada polígono.

```
public void loadPolygons() {
    int fila = 0;
    int celda;
    LatLng centro;
    String[] punto;
    //Array que contendra todos los puntos del poligono
    ArrayList<LatLng> puntos = new ArrayList<LatLng>();

    //Para cada fila de Datos
    for (List row : Datos) {
        //Limpiamos el array list
        puntos.clear();
        //Guardamos en celda la posicion del primer punto del poligono
        celda = gb.getPPOINT();
        //Obtenemos el centro
        centro = new LatLng(Double.parseDouble(row.get(gb.getLAT_POS()).toString()),
               Double.parseDouble(row.get(gb.getLNG_POS()).toString()));
        //Recorremos la Lista hasta el final obteniendo los puntos
        while (celda < row.size()) {
            //Comprobamos que existe una cadena para evitar errores
            if (row.get(celda).toString().compareTo("") != 0) {
                //Separamos la latitud de la longitud
                punto = row.get(celda).toString().split(",");
                puntos.add(new LatLng(Double.parseDouble(punto[0]),
                       Double.parseDouble(punto[1])));
            }
            celda++;
        }
        if (puntos.size() > 1) {
            createPolygon(puntos, centro, fila);
        }
        fila++;
    }
}
```

Listado 7.13: Método encargado de cargar todos los polígonos del mapa.

- Definir el método *setOnPolygonClickListener()* (Véase el Listado 7.15): de manera que se actualice el menú superior, como se hace también en los otros tipos de mapas y además, para poder mostrar la *InfoWindow* con la información del edificio que representa el polígono seleccionado, se modifica el marcador oculto de éste, de manera que se vuelva visible tanto el marcador como la ventana de información y a su vez, si se había seleccionado otro polígono, se oculta el marcador de este último.

Este tipo de mapeado es el más complejo, ya que requiere de varios puntos y por defecto, el objeto *Polygon*, no cuenta con una *InfoWindow*, siendo necesario asociar a cada polígono un marcador, para disponer de ésta.

```
1  public void createPolygon(ArrayList<LatLng> puntos, LatLng centro, int fila) {
2      //Creamos el objeto que contiene la informacion que necesitamos de cada marcador
3      MarkersData mData = new MarkersData();
4      //Guardamos las opciones del poligono
5      PolygonOptions polOptions = new PolygonOptions();
6      for (LatLng punto : puntos) {
7          polOptions.add(punto);
8      }
9
10     polOptions.strokeColor(Color.BLACK),
11              .fillColor(Color.argb(75, 50, 0, 255)),
12              .clickable(true);
13     //Creamos el poligono y se representa en el mapa
14     final Polygon poligono = getmMap().addPolygon(polOptions);
15
16     MarkerOptions markerOptions = new MarkerOptions()
17             .position(centro)
18             .title(Datos.get(fila).get(gb.getTITLE()).toString())
19             .flat(true)
20             .snippet(Datos.get(fila).get(gb.getDESC()).toString())
21             .visible(false);
22
23     Marker marcador = getmMap().addMarker(markerOptions);
24     polyMarker.put(poligono.getId(), marcador);
25     if (Datos.get(fila).size() > gb.getIMAGE() && Datos.get(fila).get(gb.getIMAGE()) != null)
26         mData.setImage(Datos.get(fila).get(gb.getIMAGE()).toString());
27     if (Datos.get(fila).size() > gb.getURL() && Datos.get(fila).get(gb.getURL()) != null)
28         mData.setUrls(Datos.get(fila).get(gb.getURL()).toString());
29
30     getMarkerData().put(marcador, mData);
31 }
```

Listado 7.14: Método encargado de crear un polígono.

```
mMap.setOnPolygonClickListener(new GoogleMap.OnPolygonClickListener() {
    @Override
    public void onPolygonClick(Polygon polygon) {
        //ocultar el ultimo marcador
        if (lastMarker != null) {
            lastMarker.hideInfoWindow();
            lastMarker.setVisible(false);
        }
        //Actualizamos como ultimo marcador el que vamos a mostrar
        lastMarker = polyMarker.get(polygon.getId());
        //Lo hacemos visible
        lastMarker.setVisible(true);
        //mostramos su infowindow asociada
        lastMarker.showInfoWindow();

        polCamUpd = CameraUpdateFactory
                .newLatLngZoom(lastMarker.getPosition(), gb.getBUILD_ZOOM());
        getmMap().moveCamera(polCamUpd);

        //Actualizamos el menu
        String[] urls = getMarkerData().get(lastMarker).getUrls().split(";");
        String[] nameAsign;
        getMenuID().findItem(R.id.topMenu).getSubMenu().clear();
        for (String url : urls) {
            nameAsign = url.split("-");
            if (nameAsign.length > 1)
                getOption_Url().put(nameAsign[0], nameAsign[1]);
            else
                getOption_Url().put(nameAsign[0], nameAsign[0]);
            getMenuID().findItem(R.id.topMenu).getSubMenu().add(Menu.NONE, 1, Menu.NONE, nameAsign
                 [0]);
        }

    }
});
```

Listado 7.15: *Listener* para los polígonos de la app.

Capítulo 8

Conclusiones y líneas de trabajo futuras

En este capítulo se presentarán las conclusiones que se pueden extraer a partir de la realización de este TFG y discutiremos posibles líneas de trabajo futuras.

8.1. Conclusiones

Durante la realización de este TFG, se han adquirido diversos conocimientos sobre una variedad de herramientas, cada una de ellas con una función concreta sobre el mismo.

Las APIs de Google no están actualmente completas para trabajar con ellas en Android, ya que fueron creadas para aplicaciones web, lo que genera diversas dificultades a la hora de trabajar con ellas, ya que a la hora de buscar información ésta aparece principalmente para aplicaciones web y no para Android, y muchas funcionalidades, que harían la aplicación más cómoda, no están disponibles.

El IDE de Android Studio ofrece muchas facilidades tanto a la hora de crear las distintas ventanas de la aplicación como a la hora de exportar al proyecto en formato de APK, permitiendo instalar en cualquier dispositivo, con los permisos adecuados, la aplicación que se desarrolla en él.

Al trabajar con LaTeX, se ha aprendido a utilizar una herramienta, que permite crear textos técnicos, lo cual aporta calidad a la hora de generar documentos. Esta herramienta permite separar las distintas partes o capítulos del documento, de manera que no es necesario tenerlo todo junto; crear variables que muestran un contenido concreto, de manera que resulte sencillo actualizar contenido; crear automáticamente el índice, tanto de los distintos capítulos como de las figuras, y la bibliografía. En definitiva ha sido una gran ayuda a la hora de elaborar esta memoria.

Finalmente, respecto a la aplicación en sí, es interesante el facilitar a los usuarios encontrar ubicaciones de un entorno concreto, en el caso de este TFG localizar edificios de La Universidad de La Laguna, ya que al visitar lugares nuevos y no conocer la zona, siempre es una ayuda el poder localizar, y disponer de información e imágenes de aquello que se busque, otorgando cierta tranquilidad. Además sería interesante poder realizar consultas desde la aplicación, de manera que al seleccionar un edificio se pudiera solicitar cita en la secretaría, consultar los libros de la biblioteca, reservar carrels, etc. otorgando facilidades y ahorrando tiempo al no tener que acceder a la web, sino que desde la aplicación se pueda realizar todos los trámites.

8.2. Conclusions

During the realization of this TFG, different knowledge has been acquired on a variety of tools, each of them with a concrete function on it.

The Google APIs are not currently complete to work with them on Android, since they were created for web applications, which generates several difficulties when working with them, since when looking for information it appears mainly for web and not for Android, and many features, which would make the application more comfortable, are not available.

The IDE of Android Studio offers many facilities both when creating the different windows of the application and when exporting the project in APK format, allowing to install it in any device, with the appropriate permissions, the application that is developed.

Working with LaTeX, you have learned how to use a tool, which allows you to create technical texts, which brings quality when generating documents. This tool allows to separate the different parts or chapters of the document, so that it is not necessary to have it all together; create variables that show a specific content, so that it is easy to update content; automatically creates the index, both the chapters and figures, and the bibliography. In short, it has been a great help in developing this memory.

Finally, regarding the application itself, it is interesting to facilitate users to find locations of a specific environment, in the case of this TFG locate buildings of La Universidad de La Laguna, since when visiting new places and not knowing the area, it is always a help being able to locate, and have information and images of what is sought, giving a certain tranquility. In addition it would be interesting to be able to make inquiries from the application, so that when selecting a building you could request an appointment in the secretariat, consult the books from the library, reserve carrels, etc. Providing facilities and saving time by not having to access the web, but from the application can perform all the procedures.

Capítulo 9

Presupuesto

En este capítulo se expondrán las estimaciones de recursos necesarios para desarrollar y publicar el proyecto.

El desglose del presupuesto de la aplicación se puede contabilizar teniendo en cuenta los siguientes aspectos:

- En caso de que se desee publicar la aplicación en la Google Play Store [11], es necesario tener una cuenta para desarrolladores de Google. Para convertir una cuenta normal en una para desarrolladores se debe acceder al siguiente enlace [12] y seguir los pasos que se indican en ella, siendo necesario un abono único de 25 dólares.

- Por defecto en Google Drive se dispone de 15GB de espacio, partiendo de que la cuenta desde la que se subiese esta app, se puede utilizar para otros proyectos podría ser necesario aumentar el espacio disponible, para lo que Google ofrece varias cuotas:

  - 100GB: 1,99 euros al mes o 19,99 euros al año.
  - 1TB: 9,99 euros al mes o 99,99 euros al año.
  - 2TB: 19,99 euros al mes.
  - 10TB: 99,99 euros al mes.
  - 20TB: 199,99 euros al mes.
  - 30TB: 299,99 euros al mes.

  En principio no es necesario mejorar el espacio disponible para la aplicación, pero en el futuro podría ser necesario al añadir nuevas funcionalidades.

- Es importante que la aplicación muestre un diseño agradable al usuario y que a la vez represente a la compañia que la ofrece, se estima necesario

los servicios de un diseñador para obtener buenos resultados, lo cual tiene un coste aproximado de entre 200-400 euros mensuales dependiendo del diseñador y el tiempo que le consuma.

- Los datos que requiere la aplicación de cada edificio, es importante que se obtengan de primera mano, de manera que la información que muestre a los usuarios sea precisa, por lo que lo ideal es ir visitando cada centro, comprobando los servicios que oferta y añadir cualquier dato relevante que se encuentre durante la visita y además obtener fotos para la app. Para poder realizarse esta investigación es necesario visitar las distintas facultades, institutos, colegios mayores y cualquier otro edificio relacionado con la universidad, por lo que habría que incluir gastos de transporte e investigación, aproximadamente 400-600 euros.

- El desarrollo de la aplicación conlleva un número de horas que tendrá que desempeñar un programador. Esto supone un costo que puede contemplarse de dos maneras distintas:

  - Pago fijo: se indica un pago único al programador por la labor de desarrollo. Para el caso de `ULL-Maps` se considera que supondría un gasto de 1500 euros.
  - Pago por horas: se calcula el pago del programador según las horas que haya invertido en la realización del trabajo, dejando claro cuánto cobrará por cada hora. En el caso de la aplicación `ULL-Maps` se considera un pago de 40 euros/hora. El precio en este caso puede variar mucho dependiendo del programador y lo claro que este el proyecto que deba realizar.

Considerando los datos aportados para el presupuesto y el pago fijo para el programador, ya que por horas es un dato menos concreto, obtenemos un costo mínimo de 2125 euros y un máximo de 2525 euros más el costo mensual de los trabajadores en caso de que fuera necesario lo que supone una suma de entre 240 y 440 euros más por mes.

Bibliografía

[1] **Android** 2008, [https://www.android.com/]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 2

[2] Gitlab: **Repositorio Gitlab** 2011, [https://gitlab.com/]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 2, 4

[3] LaTeX3 Project: **LaTeX** 1985, [https://www.latex-project.org/]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 2, 4

[4] IntelliJ: **Android Studio** 2014, [https://developer.android.com/studio/index.html]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 3, 14, 20, 28

[5] **IntelliJ IDEA** 2001, [https://www.jetbrains.com/idea/]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 3

[6] **Gradle** 2007, [http://gradle.org/]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 3

[7] **API Google Maps**[https://developers.google.com/maps/documentation/android-api]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 5, 15, 33, 45

[8] **API Google Sheet**[https://developers.google.com/sheets/api]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 5, 15

[9] **Google Developer Console**[https://console.developers.google.com/apis/dashboard?]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 15

[10] **SpreadSheet utilizado para la App** ULL-Maps 2017, [http://tinyurl.com/ULLMaps/]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 45

[11] **Google Play** [https://play.google.com/store]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 51

[12] **Desde aqui se actualiza una cuenta de Google normal a desarrollador** 2017, [https://play.google.com/apps/publish/signup/]. [[Disponible electrónicamente. Último acceso, junio de 2017]]. 51