Localización geográfica en Android

Veremos cómo acceder a los datos de localización o ubicación actual del dispositivo en el que se está ejecutando la aplicación Android. El contar con datos de ubicación va a multiplicar las posibilidades a la hora de crear nuestras aplicaciones, permitiendo ofrecer al usuario información relativa al *contexto* en el que se encuentra.

Desde el punto de vista del desarrollador, hay que decir que la API de ubicación es una de esas características de Android que quizá no sean tan intuitivas como deberían ser, por lo que es importante estar atento a los detalles, aunque bien es cierto que ha ido mejorando con los años y a día de hoy contamos con muchas "facilidades" que antes no teníamos. Como ejemplo más llamativo, actualmente no tenemos que preocuparnos demasiado, al menos no de forma explícita, de qué proveedor de localización queremos utilizar en cada momento (señal móvil, Wi-Fi, o GPS), ya que tenemos disponible un nuevo proveedor (*Fused Location Provider*) que se encargará automáticamente de gestionar todas las fuentes de datos disponibles para obtener la información que nuestra aplicación necesita.

La funcionalidad de localización geográfica fue movida hace ya algún tiempo desde el SDK general de Android a las librerías de los *Google Play Services.* Por lo tanto, lo primero que tendremos que aprender será a conectarnos a dichos servicios.

El primer paso, siempre que queramos hacer uso de los Google Play Services, será añadir a la sección de dependencias de nuestro fichero *build.gradle* la referencia a las librerías necesarias. En el caso de estos servicios existen dos alternativas, o bien añadimos la referencia a la **librería completa de Google Play Services**, que nos daría acceso a todos los servicios disponibles, o bien utilizamos las **librerías independientes de cada servicio** que vayamos a utilizar.

Para la primera opción la referencia a añadir sería la siguiente:

```
. . .
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
    implementation 'com.google.android.gms:play-services:12.0.1'
}
```

Y si sólo queremos añadir la **librería de localización** (que también incluye las funciones de *Reconocimiento de Actividad* y *Google Places*), utilizaríamos la siguiente:

```
. . .
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
```

```
    implementation 'com.google.android.gms:play-services-location:16.0.0'
}
```

Por norma general utilizaremos esta segunda alternativa, añadiendo únicamente las librerías de los servicios que vamos a utilizar. Se puede encontrar el listado completo de librerías para los servicios disponibles en el siguiente enlace.

Hecho esto vamos a empezar a crear una aplicación de ejemplo. No nos complicaremos mucho, incluiremos tan sólo un par de etiquetas de texto para mostrar la latitud y longitud recibidas, y un botón para iniciar o parar las actualizaciones de posición.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical">

    <TextView android:id="@+id/lblLatitud"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/latitud"/>

    <TextView android:id="@+id/lblLongitud"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/longitud"/>

    <ToggleButton android:id="@+id/btnActualizar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textOn="@string/parar_actualizaciones"
        android:textOff="@string/iniciar_actualizaciones" />

</LinearLayout>
```

Obtenemos como siempre las referencias a estos controles en el método `onCreate()` de la actividad principal, y crearemos un método auxiliar `updateUI()` que actualice los campos de latitud y longitud a partir de un objeto `Location`, que como veremos más adelante será lo que obtengamos del servicio de localización.

```
...
import android.location.Location;

//...

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

        tvLatitud = (TextView) findViewById(R.id.lblLatitud);
        tvLongitud = (TextView) findViewById(R.id.lblLongitud);
```

```
        btnActualizar = (ToggleButton) findViewById(R.id.btnActualizar);

//...

    }

    private void updateUI(Location loc) {
        if (loc != null) {
            tvLatitud.setText("Latitud: " + String.valueOf(loc.getLatitude()));
            tvLongitud.setText("Longitud: " + String.valueOf(loc.getLongitude()));
        } else {
            tvLatitud.setText("Latitud: (desconocida)");
            tvLongitud.setText("Longitud: (desconocida)");
        }
    }
```

El objeto **`Location`** simplemente **encapsula los datos principales de una dirección geográfica, entre otros la latitud y longitud**, a los que podemos acceder mediante los métodos `getLatitude()` y `getLongitude()` respectivamente. En caso de recibirse un objeto nulo mostraremos un literal de dirección "*(desconocida)"*, más adelante veremos por qué puede ocurrir esto.

Hechos los preparativos iniciales, vamos a añadir funcionalidad más específica. Comenzaremos por crear un objeto de tipo `GoogleApiClient`, que generalmente será el punto de acceso común a los servicios de *Google Play*. Para la creación de este objeto debemos indicar la API a la que queremos acceder y cómo queremos gestionar la conexión con los servicios (de forma manual o automática, utilizaremos la segunda opción siempre que sea posible). Habrá que proporcionar también los *listeners* necesarios para responder a los eventos de conexión/desconexión a los servicios y a posibles errores que pudieran producirse durante la conexión. Veamos en primer lugar el código de creación de este objeto y posteriormente pasaremos a comentar los detalles.

```
GoogleApiClient apiClient = new GoogleApiClient.Builder(this)
        .enableAutoManage(this, this)
        .addConnectionCallbacks(this)
        .addApi(LocationServices.API)
        .build();
```

Como puede comprobarse, la configuración del cliente se realiza utilizando un patrón *builder* a través de la clase `GoogleApiClient.Builder`.

En primer lugar solicitamos con `enableAutoManage()` que la gestión de la conexión a los servicios se realice automáticamente, esto nos evitará entre otras cosas tener que conectarnos y desconectarnos manualmente a los servicios en los eventos `onStart()` y `onStop()` de la actividad, y además también se gestionarán de forma automática muchos de los posibles errores que puedan producirse durante la conexión (por ejemplo, solicitando al usuario una actualización de los *play services* en el dispositivo, si fuera necesario). Este método recibe dos parámetros, el primero de ellos es una referencia a la actividad en cuyo ciclo de vida debe integrarse la conexión/desconexión a los servicios, en nuestro caso será la propia actividad principal (`this`). El segundo parámetro es la referencia al listener (de tipo `OnConnectionFailedListener`) que se llamará en caso de producirse algún error que el sistema no pueda gestionar automáticamente. En nuestro caso haremos que la propia actividad principal implemente la interfaz `OnConnectionFailedListener`, por lo que también pasaremos `this` en este segundo parámetro. Para implementar esta interfaz

tendremos que definir el método `onConnectionFailed(ConnectionResult)`, que en nuestro caso de ejemplo se limitará a mostrar un mensaje de error en el log:

```
@Override
public void onConnectionFailed(ConnectionResult result) {
    //Se ha producido un error que no se puede resolver automáticamente
    //y la conexión con los Google Play Services no se ha establecido.

    Log.e(LOGTAG, "Error grave al conectar con Google Play Services");
}
```

Lo siguiente que hacemos es indicar el listener que se ocupará de responder a los eventos de conexión y desconexión de los servicios, mediante una llamada a `addConnectionCallbacks()`. Esto no es obligatorio, aunque en la práctica casi siempre nos interesará conocer cuándo se realiza la conexión y cuándo se pierde, de forma que podamos adaptarnos convenientemente a cada situación (es importante entender que no podemos hacer llamadas a los servicios mientras no estemos conectados a ellos). Una vez más haremos que nuestra actividad implemente la interfaz necesaria, en este caso `ConnectionCallbacks`, lo que nos obligará a implementar los métodos `onConnected()`, que se ejecutará cuándo se realice la conexión con los servicios, y `onConnectionSuspended()`, que se lanzará cuando la conexión se pierda temporalmente (cuando la conexión se recupera volverá a lanzarse el evento `onConnected()`).

Posteriormente indicamos mediante `addApi()` la API de los servicios a los que vamos a acceder, en este caso `Location.API`. En caso de querer utilizar diferentes servicios podríamos realizar varias llamadas al método `addApi()` añadiendo los servicios necesarios.

Por último, construimos el objeto final llamando al método `build()`. Esto iniciará la conexión con los servicios solicitados, lo que desembocará como hemos indicado en una llamada al evento `onConnectionFailed()` en caso de error, o al evento `onConnected()` en el caso de funcionar todo correctamente.

El evento `onConnected()` se aprovecha frecuentemente para realizar la interacción deseada con el servicio, una vez que estamos seguros que la conexión se ha realizado correctamente. En nuestro caso utilizaremos este evento para obtener una primera posición inicial para inicializar los datos de latitud y longitud de la interfaz. Para ello obtendremos la *última posición geográfica conocida*, lo que conseguimos llamando al método `getLastLocation()` del proveedor de datos de localización `LocationServices.FusedLocationApi`, pasándole como parámetro la referencia al cliente API que hemos creado anteriormente. Por último llamamos a nuestro método auxiliar `updateUI()` con el valor recibido para mostrar los datos en la actividad principal. Vemos a continuación el evento `onConnectionSuspended()`, que para nuestro ejemplo se limitará a mostrar un mensaje en el log.

```
@Override
public void onConnected(@Nullable Bundle bundle) {
    //Conectado correctamente a Google Play Services

 ...

     Location lastLocation =
             LocationServices.FusedLocationApi.getLastLocation(apiClient);

      updateUI(lastLocation);
```

```
        ...

    }

    @Override
    public void onConnectionSuspended(int i) {
        //Se ha interrumpido la conexión con Google Play Services

        Log.e(LOGTAG, "Se ha interrumpido la conexión con Google Play
    Services");
    }
```

¿Por qué hablamos de "*última posición conocida*" y no de "*posición actual*"? En Android no existe ningún método del tipo "*obtenerPosiciónActual()*". Obtener la posición a través de un dispositivo de localización como por ejemplo el GPS no es una tarea inmediata, sino que puede requerir de un cierto tiempo de espera y procesamiento, por lo que no tendría demasiado sentido proporcionar un método de ese tipo. Lo más parecido que encontramos es el método `getLastLocation()` que nos devuelve la posición más reciente obtenida por el dispositivo (no necesariamente solicitada por nuestra aplicación). Y es importante entender que: este método **NO devuelve la posición actual**, este método **NO solicita una nueva posición al proveedor de localización**, este método **se limita a devolver la última posición que se obtuvo a través del proveedor de localización**. Y esta posición se pudo obtener hace unos pocos segundos, hace días, hace meses, o incluso nunca (si el dispositivo ha estado apagado, si nunca se ha activado el GPS, …). Por tanto, cuidado cuando se haga uso de la posición devuelta por el método `getLastLocation()`. En los casos raros en que no existe ninguna posición conocida en el dispositivo este método devuelve `null`, algo para lo que ya hemos preparado nuestro método `updateUI()`.

En el código anterior del evento `onConnected()` se ha omitido inicialmente, por claridad, un fragmento de código relacionado con la obtención de los permisos necesarios para que la aplicación pueda acceder a datos de localización.

Para empezar, indicar que los permisos relacionados con la ubicación geográfica en Android son básicamente dos:

- **`ACCESS_FINE_LOCATION`**. Permiso para acceder a datos de localización con una precisión alta.
- **`ACCESS_COARSE_LOCATION`**. Permiso para acceder a datos de localización con una precisión baja.

Dependiendo de qué permiso/s solicita nuestra aplicación, los datos de localización obtenidos posteriormente podrán tener una mayor o menor precisión.

Para versiones de Android hasta la 5.0 los permisos necesarios para la aplicación se deben declarar en el fichero `AndroidManifest.xml`, y deben ser aceptados por el usuario (todos o ninguno) en el momento de instalar la aplicación. En nuestro caso solicitaremos permisos para obtener ubicaciones con la máxima precisión disponible, añadiendo la cláusula correspondiente `<uses-permission>` a nuestro fichero *AndroidManifest.xml.*

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.ejlocalizacionmapas">
```

```
    <uses-permission
        android:name="android.permission.ACCESS_FINE_LOCATION" >
    </uses-permission>
    . . .
</manifest>
```

Sin embargo, a partir de Android 6.0 algunos permisos se deben consultar en tiempo de ejecución, con lo que el usuario decidirá si conceder o no, cada uno de ellos, en el momento en que se haga uso de cada funcionalidad que los necesite (y no en la instalación como ocurría en versiones anteriores). Nuestra aplicación debería estar preparada por tanto para todas las situaciones posibles, es decir, debería funcionar sin errores tanto si se le conceden los permisos solicitados como si no (deshabilitando por supuesto en este caso la funcionalidad asociada).

Recordamos rápidamente cómo chequear y solicitar permisos en tiempo de ejecución para versiones de *Android 6.0 o superior*. Para chequear si nuestra aplicación tiene concedido un determinado permiso utilizamos el método `checkSelfPermission()`. En el caso de no tenerlo aún concedido, lo solicitaremos al usuario llamado a `requestPermissions()`, pasándole como parámetro el identificador del permiso deseado y una constante arbitraria definida por nuestra aplicación (`PETICION_PERMISO_LOCALIZACION` en el ejemplo) que nos permita identificar posteriormente dicha petición. Para conocer el resultado de la petición sobrescribiremos el evento `onRequestPermissionResult()`, utilizando la constante indicada para reconocer a qué petición corresponde el resultado recibido.

A continuación completamos el código del evento `onConnected()` con el chequeo/petición de permisos y añadimos la implementación de `onRequestPermissionResult()` para actuar según la respuesta del usuario a la petición de permisos en caso de haberse realizado.

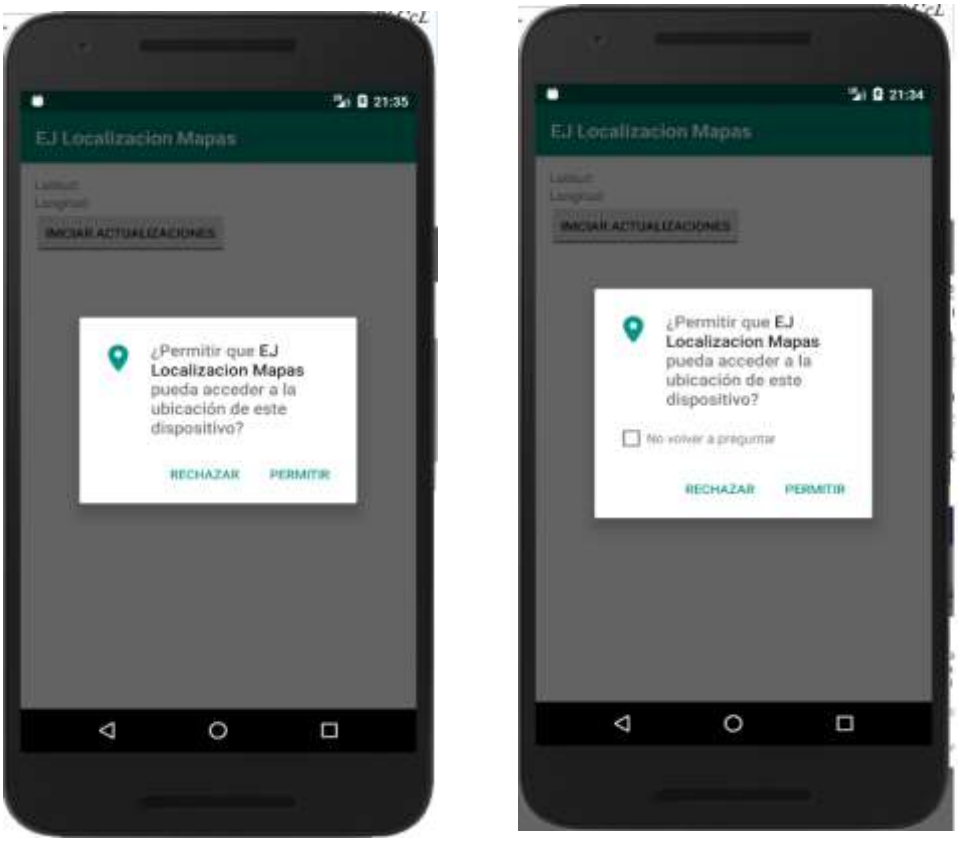
```
@Override
public void onConnected(@Nullable Bundle bundle) {
    //Conectado correctamente Google Play Services

    //Comprobamos si tenemos permisos
    if (ActivityCompat.checkSelfPermission(this,
            Manifest.permission.ACCESS_FINE_LOCATION) !=
                                PackageManager.PERMISSION_GRANTED) {

        //No tenemos permiso, lo solicitamos.
        Log.e("LOGTAG", "VAMOS A PEDIR PERMISO");
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
                PETICION_PERMISO_LOCALIZACION);
    } else {
        //Obtenemos la ultima posicón conocida
        inicializarLocalizacion();
    }
}

private void inicializarLocalizacion () {
    try {
        Location lastLocation =
               LocationServices.FusedLocationApi.getLastLocation(apiClient);
        updateUI(lastLocation);
    } catch (SecurityException se) {
```

```
            se.getMessage();
        }
    }


    @Override
    public void onRequestPermissionsResult(int requestCode,
                                           @NonNull String[] permissions,
                                           @NonNull int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);

        if (requestCode == PETICION_PERMISO_LOCALIZACION) {
            if (grantResults.length == 1 &&
                    grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                // Permiso concedido. Obtenemos la ultima posicón conocida
                Log.e("LOGTAG", "PERMISOOOOOO CONCEDIDO");
                inicializarLocalizacion ();
            }
            else {
                //Permiso denegado.
                //Deberiamos deshabilitar toda la funcionalidad relativa
                //a la localización
                Log.e("LOGTAG", "Permiso denegado");
            }
        }

    }
```

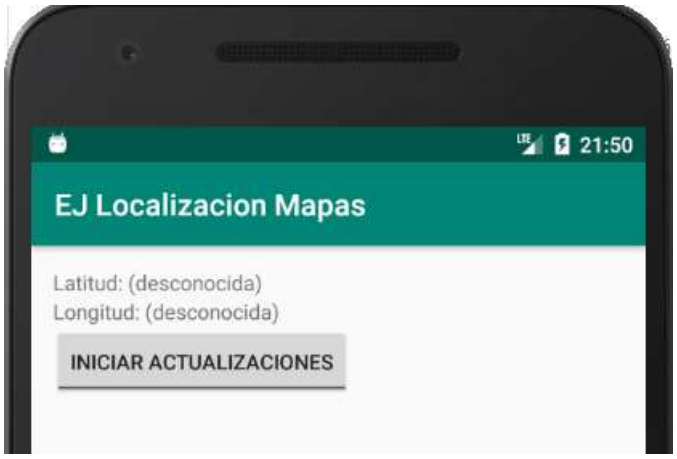
Para nuestro ejemplo, en el evento `onRequestPermissionResult()`, en el caso de obtener los permisos solicitados hacemos exactamente lo mismo que en el evento `onConnected()`, es decir, obtener la última posición conocida y actualizar la interfaz. En caso de no recibir los permisos por parte del usuario mostramos simplemente un mensaje de error en el log.

Con esto, ya tendríamos una versión inicial, muy básica, de la aplicación de ejemplo, que al iniciarse obtendría y mostraría la última posición recibida en el dispositivo.

Si ejecutamos en este momento la aplicación en el emulador, lo primero que deberíamos ver es la petición del permiso para acceder a la ubicación:



Si pulsamos en el botón *PERMITIR* del diálogo para aceptar el permiso solicitado por la aplicación pueden pasar dos cosas. Si la aplicación se está ejecutando en un emulador recién creado, o donde nunca se ha ejecutado ninguna aplicación que acceda a ubicaciones es muy posible que obtengamos una localización nula, por lo que se mostraría el literal "*(desconocida)*" en los valores de latitud y longitud.



Si por el contrario ya se había obtenido alguna ubicación en el emulador, o bien si estamos ejecutando la aplicación sobre un dispositivo físico, deberían mostrarse sin problemas los valores correspondientes de latitud y longitud de la ubicación más reciente recibida.

Ahora, vamos a activar la recepción periódica de ubicaciones (mediante el botón "*Iniciar Actualizaciones"* que hemos incluido en la interfaz) de forma que la aplicación esté periódicamente recibiendo, esta vez sí, la posición real y exacta del dispositivo.

Vamos a describir cómo solicitar al sistema datos actualizados, esta vez sí, de la posición actual del dispositivo, por supuesto siempre cumpliendo con el nivel de permisos y precisión que hayamos solicitado.

Como ya hemos comentado, en Android no tenemos ningún método que nos devuelva directamente la posición actual, entre otras cosas porque es impredecible el tiempo que podemos tardar en obtenerla. Seguiremos por tanto otra estrategia, que consistirá en primer lugar en indicar al sistema nuestros *requerimientos*, entre ellos la precisión y periodicidad con que nos gustaría recibir actualizaciones de la posición actual, y en segundo lugar definiremos un método encargado de procesar los nuevos datos a medida que se vayan recibiendo.

Antes de esto otra cosa importante. En la precisión de los datos obtenidos no solo interviene lo que nuestra aplicación solicite, sino también la configuración del dispositivo que el usuario tenga establecida. Por ejemplo, nuestra aplicación no podría obtener, aunque así lo solicite, una ubicación con máxima precisión si el usuario lleva deshabilitada la *Ubicación* en el dispositivo, o si el modo que tiene seleccionado en las opciones de ubicación de Android no es el de "*Alta precisión*". Por tanto, un primer paso importante será comprobar de alguna forma si las necesidades de nuestra aplicación son coherentes con la configuración actual establecida en el dispositivo, y en caso contrario solicitar al usuario que la modifique siempre que sea posible.

Vayamos paso a paso. La forma en que nuestra aplicación puede definir sus requerimientos en cuanto a opciones de ubicación será a través de un objeto de tipo `LocationRequest`. Este objeto almacenará las opciones de ubicación que nuestra aplicación necesita, entre las que destacan:

- **Periodicidad de actualizaciones**. Se establece mediante el método `setInterval()` y define cada cuanto tiempo (en milisegundos) *nos gustaría* recibir datos actualizados de la posición. De esta forma, si queremos recibir la nueva posición cada 2 segundos utilizaremos `setInterval(2000)`. Y decimos "nos gustaría" porque con este método lo que damos es nuestra preferencia, pero la periodicidad real podría ser mayor o menor dependiendo de muchas circunstancias (conectividad GPS limitada o intermitente, otras aplicaciones han solicitado periodicidades más altas, …).

- **Periodicidad máxima de actualizaciones**. El proveedor de localización de Android (*Fused Location Provider*) proporciona actualizaciones de la ubicación con la periodicidad más alta que haya solicitado *cualquier* aplicación ejecutándose en el dispositivo (éste es uno de los motivos por los que en el apartado anterior indicábamos que es posible recibir actualizaciones a mayor velocidad de la solicitada). Por este motivo, es importante indicar al sistema a qué periodicidad máxima (también en milisegundos) nuestra aplicación es capaz de procesar nuevos datos de ubicación de forma que no nos provoque problemas de rendimiento o sobrecarga. Este dato lo proporcionaremos mediante el método `setFastestInterval()`.
- **Precisión**. La precisión de los datos que queremos recibir se establecerá mediante el método `setPriority()`. Existen varios valores posibles para definir esta información:
    - `PRIORITY_BALANCED_POWER_ACCURACY`. Los datos recibidos tendrán una precisión de unos 100 metros. En este modo el dispositivo tendrá un consumo de energía comedido al utilizar normalmente la señal WIFI y de datos móviles para determinar la ubicación.
    - `PRIORITY_HIGH_ACCURACY`. Es el modo más preciso para obtener la ubicación, por lo que utilizará normalmente la señal GPS.
    - `PRIORITY_LOW_POWER`. Los datos recibidos tendrán una precisión de unos 10 kilómetros, pero se utilizará muy poca energía para obtener la ubicación.
    - `PRIORITY_NO_POWER`. En este modo nuestra aplicación solo recibirá datos si éstos están disponibles porque *alguna otra aplicación* los haya solicitado. Es decir, nuestra aplicación no tendrá un impacto directo en el consumo de energía solicitando nuevas ubicaciones, pero si éstas están disponibles las utilizará.

Con esta información vamos a definir ya el `LocationRequest` para el ejemplo. Crearemos un método auxiliar `enableLocationUpdates()` al que llamaremos desde nuestro botón de iniciar/detener las actualizaciones. Para el ejemplo utilizaremos una periodicidad de 2 segundos, una periodicidad máxima de 1 segundo, y una precisión alta (`PRIORITY_HIGH_ACCURACY`).

```
private LocationRequest locRequest;

 ...

@Override
protected void onCreate(Bundle savedInstanceState) {

 ...

    btnActualizar = (ToggleButton) findViewById(R.id.btnActualizar);
    btnActualizar.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            toggleLocationUpdates(btnActualizar.isChecked());
        }
    });


 ...

}
```

```
private void toggleLocationUpdates(boolean checked) {
    if (checked)
        enableLocationUpdates();
    else
        disableLocationUpdates();
}

private void enableLocationUpdates() {

    locRequest =new LocationRequest();
    locRequest.setInterval(2000);
    locRequest.setFastestInterval(1000);
    locRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);

 ...

}
```

Definidos nuestros requisitos vamos ahora a comprobar si la configuración actual del dispositivo es coherente con ellos. Para ello construiremos, a continuación dentro de `enableLocationUpdates()`, un objeto `LocationSettingsRequest` mediante su *builder*, al que pasaremos el `LocationRequest` definido en el paso anterior.

```
LocationSettingsRequest locSettingsRequest =
                        new LocationSettingsRequest.Builder()
                            .addLocationRequest(locRequest)
                            .build();
```

Con esto queremos que de alguna forma el sistema compare los requisitos de nuestra aplicación con la configuración actual. ¿Pero cómo ejecutamos y conocemos el resultado de dicha comparación? Para esto llamaremos al método `checkLocationSettings()` de la API de localización, al que pasaremos la instancia de nuestro cliente API y del `LocationSettingsRequest` que acabamos de construir. El resultado vendrá dado en forma de objeto `PendingResult`, del que tendremos de definir su evento `onResult()` para conocer el resultado de la comparación una vez esté disponible. Este evento recibe como parámetro un objeto `LocationSettingsResult`, cuyo método `getStatus()` contiene el resultado de la comparación.

Contemplaremos tres posibles resultados:

- `SUCCESS`. Significará que la configuración del dispositivo es válida para nuestros requisitos de información.
- `RESOLUTION_REQUIRED`. Indica que la configuración actual del dispositivo no es suficiente para nuestra aplicación, pero existe una posible solución por parte del usuario (por ejemplo: solicitarle que active la ubicación en el dispositivo o que cambie su modalidad).
- `SETTINGS_CHANGE_UNAVAILABLE`. Indica que la configuración del dispositivo no es suficiente y además no existe ninguna acción del usuario que pueda solucionarlo.

En el primer caso ya podríamos solicitar el inicio de las actualizaciones de localización, ya que sabemos que la configuración del dispositivo es correcta. En el tercer caso, no nos quedaría más opción que mostrar algún mensaje al usuario indicando que no es posible obtener la ubicación, o bien deshabilitar la funcionalidad relacionada.

Y el caso más interesante, el segundo, necesitamos solicitar al usuario que cambie la configuración del sistema. Por suerte, esta solicitud está ya implementada en la api, por lo que tan sólo tendremos que llamar al método `startResolutionForResult()` sobre el estado recibido en el evento `onResult()`. Este método recibe una referencia a la actividad principal y una constante arbitraria (que podemos definir con cualquier valor único) que después nos servirá para obtener el resultado de la operación.

Veamos todo lo anterior sobre el código.

```
private void enableLocationUpdates() {

    locRequest =new LocationRequest();
    locRequest.setInterval(2000);
    locRequest.setFastestInterval(1000);
    locRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);

    LocationSettingsRequest locSettingsRequest =
            new LocationSettingsRequest.Builder()
                    .addLocationRequest(locRequest)
                    .build();

    PendingResult<LocationSettingsResult>result =
            LocationServices.SettingsApi.checkLocationSettings(
                    apiClient,locSettingsRequest);

    result.setResultCallback(new ResultCallback<LocationSettingsResult>() {
        @Override
        public void onResult(LocationSettingsResult locationSettingsResult) {
            final Status status = locationSettingsResult.getStatus();
            switch (status.getStatusCode()){
                case LocationSettingsStatusCodes.SUCCESS:

                    Log.i (LOGTAG,"Configuración correcta");
                    startLocationUpdates();
                    break;
                case LocationSettingsStatusCodes.RESOLUTION_REQUIRED:
                    try {
                        Log.i(LOGTAG, "Se requiere actuación del usuario");
                        status.startResolutionForResult(MainActivity.this,
                                        PETICION_CONFIG_UBICACION);
                    } catch (IntentSender.SendIntentException e) {
                        btnActualizar.setChecked(false);
                        Log.i(LOGTAG, "Error al intentar solucionar
                                            configuración de ubicación");
                    }
                    break;
                case LocationSettingsStatusCodes.SETTINGS_CHANGE_UNAVAILABLE:
                    Log.i(LOGTAG, "No se puede cumplir la configuración
                                            de ubicación necesaria");
                    btnActualizar.setChecked(false);
                    break;
            }
        }
    });
}
```

Nos faltaría saber el resultado de la solicitud realizada al usuario para cambiar la configuración en el caso de `RESOLUTION_REQUIRED`. Para ello sobrescribiremos el método `onActivityResult()` de la actividad principal, y atenderemos el caso en el que el *requestCode* recibido sea igual a la constante que utilizamos en el método `startResolutionForResult()`. Existen dos posibles resultados:

- `RESULT_OK`. Indica que el usuario ha realizado el cambio solicitado. En este caso ya podremos solicitar el inicio de las actualizaciones de ubicación.
- `RESULT_CANCELED`. Indica que el usuario no ha realizado ningún cambio. En nuestro caso de ejemplo mostraremos un error en el log y desactivaremos el botón de inicio de las actualizaciones.

```
@Override
protected void onActivityResult(int requestCode,
                        int resultCode, Intent data) {
    switch (requestCode) {
        case PETICION_CONFIG_UBICACION:
            switch (resultCode) {
                case Activity.RESULT_OK:
                    startLocationUpdates();
                    break;
                case Activity.RESULT_CANCELED:
                    Log.i(LOGTAG, "El usuario no ha realizado los
                                cambios de configuración necesarios");
                    btnActualizar.setChecked(false);
                    break;
            }
            break;
    }
}
```

Pues bien, después de todo esto, ya nos quedaría únicamente saber cómo solicitar el inicio de las actualizaciones de localización del dispositivo. Esto lo haremos en un método auxiliar `startLocationUpdates()`. Esta acción es muy sencilla, basta con llamar al método `requestLocationUpdates()` de la API de localización, pasándole como parámetros nuestro cliente API, el objeto `LocationRequest` construido al inicio y una referencia al objeto que implementará la interfaz `LocationListener`, cuyo método `onLocationChanged()` recibirá los datos de ubicación actualizados. En nuestro caso, haremos que sea nuestra actividad principal la que implemente esta interfaz. Definiremos por tanto el evento indicado en nuestra actividad, que se limitará a llamar a nuestro método auxiliar `updateUI()` con los nuevos datos recibidos.

```
public class LocalizacionActivity extends AppCompatActivity
                    implements GoogleApiClient.OnConnectionFailedListener,
                               GoogleApiClient.ConnectionCallbacks,
                               LocationListener {

    . . .

    private void startLocationUpdates() {
        if (ActivityCompat.checkSelfPermission(LocalizacionActivity.this,
                Manifest.permission.ACCESS_FINE_LOCATION) ==
                                PackageManager.PERMISSION_GRANTED) {
```

```
                //Ojo: Estamos suponiendo que ya tenemos concedio el permiso.
                //Seria recomendable implementar la posible petición
                //en caso de no tenerlo.

                Log.i ("LOGTAG", "Inicio de recepción de ubicaciones");
                LocationServices.FusedLocationApi.requestLocationUpdates(
                                    apiClient,
                                    locRequest,
                                    (LocationListener) LocalizacionActivity.this);
            }
        }

       . . .

        @Override
        public void onLocationChanged(Location location) {
            Log.i("LOGTAG", "Recibida nueva ubicación");

            //Mostramos la nueva ubicación
            updateUI(location);
        }
    }
```

Por último, para detener la actualización de ubicaciones, tan sólo tendremos que llamar al método `removeLocationUpdates()` de la API de localización, lo que haremos en un método auxiliar `disableLocationUpdates()` que a su vez llamaremos cuando corresponda al pulsar el botón de iniciar/detener actualizaciones.

```
    private void disableLocationUpdates() {

        LocationServices.FusedLocationApi.removeLocationUpdates(
                apiClient, this);

    }
```

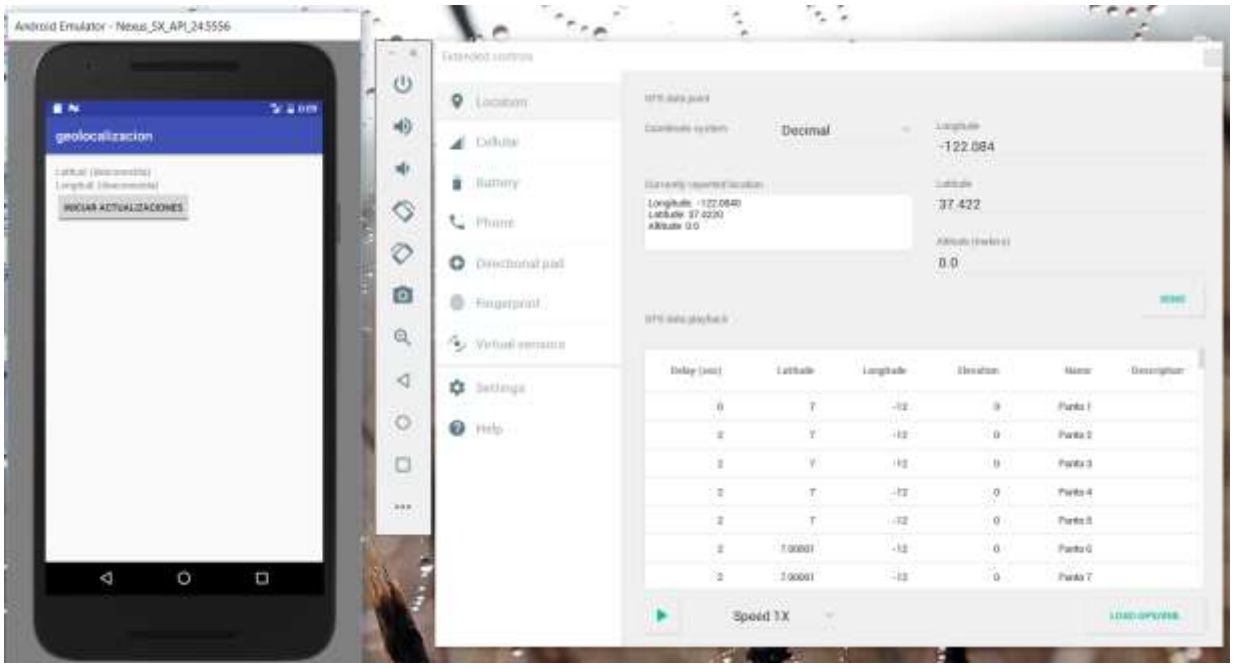
Ya estaríamos listos para ejecutar la aplicación de ejemplo en el emulador o en un dispositivo real. Para ello, configuraremos primero el dispositivo para desactivar las opciones de ubicación y poder comprobar si nuestra aplicación detecta correctamente esta situación.

Ejecutamos ahora la aplicación y aceptamos los permisos de localización si se nos solicita (en Android 6 o superior). Pulsamos el botón de "*INICIAR ACTUALIZACIONES*" y deberíamos ver un diálogo como el siguiente, donde se nos indica que debemos habilitar las opciones de Ubicación para usar la señal móvil, Wi-Fi y GPS (recordemos que hemos solicitado ubicaciones con la máxima precisión).

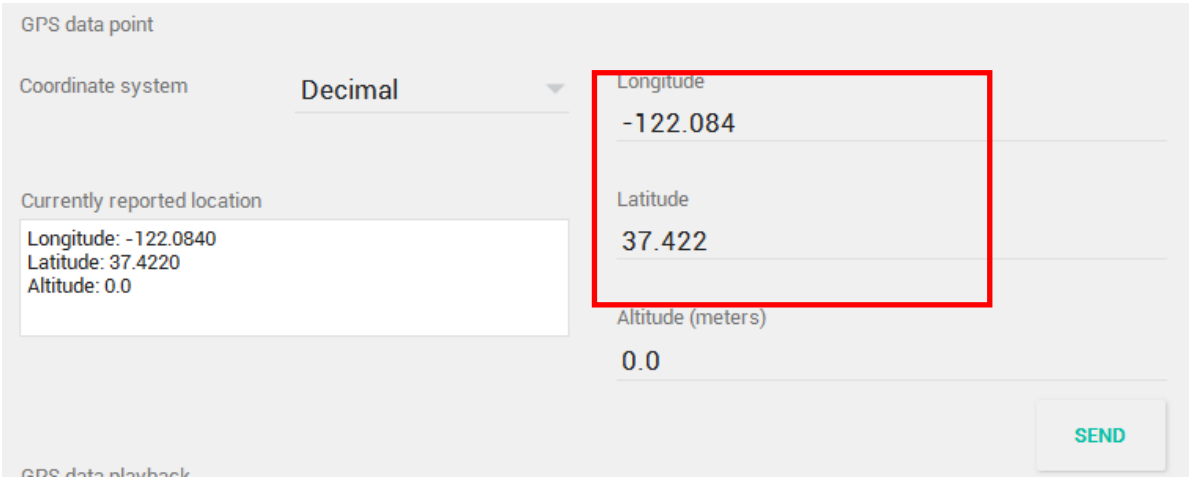
Si pulsamos “SÍ” se habilitarán automáticamente estas opciones (Ubicación activada y modo de “*Alta precisión*“) y nuestra aplicación debería comenzar a recibir actualizaciones de ubicación tal y como habíamos previsto.

Sin embargo, si estamos ejecutando la aplicación en un emulador no veremos ningún cambio en la ubicación. Latitud y Longitud quedarán con valor fijo (última posición conocida) o bien con valor “(desconocido)”. ¿Por qué ocurre esto? Muy sencillo, el emulador, al no ser un dispositivo real, no recibe señal móvil ni GPS, por lo que es incapaz de obtener la ubicación actual a partir de dicha información.

Por suerte, el emulador ofrece un método alternativo para simular que el dispositivo recibe actualizaciones de ubicación. Estas opciones se pueden encontrar accediendo a los controles extendidos del emulador, en la sección “*Location*“.



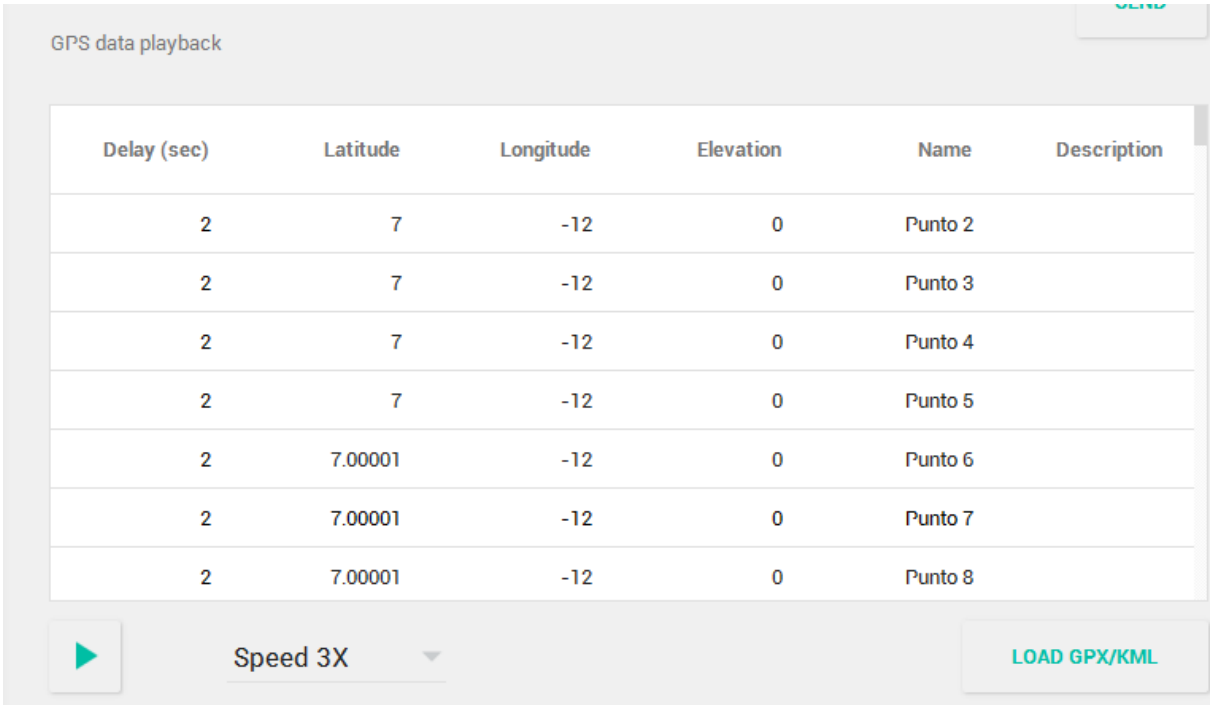
Accediendo a estos controles tendremos dos alternativas para enviar ubicaciones a nuestro emulador, una manual y otra automática. La manual, situada en la parte superior, nos permite introducir un valor de latitud-longitud y enviarlo al emulador mediante el botón “SEND”, de uno en uno. Si lo hacemos mientras nuestra aplicación se está ejecutando y nuestro botón de actualizaciones está activado, veremos cómo el dato de latitud-longitud introducido en las opciones del emulador aparece en nuestra aplicación (es posible que tarde un poco en aparecer, en general los controles extendidos del emulador van relativamente lentos dependiendo de los recursos del equipo de trabajo).



Este método, aunque efectivo, es algo laborioso si queremos probar que nuestra aplicación recibe actualizaciones de la ubicación con cierta frecuencia. Para solucionar esto podemos utilizar la segunda de las opciones, que nos permite automatizar el envío al emulador de un listado de ubicaciones a una cierta velocidad.

El listado de ubicaciones debe estar en formato GPX o KML. Para el ejemplo, vamos a probar con un archivo KML por su simplicidad. Desde este enlace podéis descargar un fichero KML de ejemplo, que podéis abrir/editar con cualquier editor de texto para adaptarlo a vuestras necesidades. Como podéis comprobar no es más que un listado de pares de latitud-longitud con una estructura muy sencilla de etiquetas tipo XML.

Para cargar este fichero pulsaremos sobre el botón inferior “LOAD GPX/KML” y seleccionaremos nuestro fichero de prueba. Inmediatamente (o no tan inmediato) aparecerán en la lista superior nuestro listado de ubicaciones. A continuación seleccionaremos la velocidad a la que queremos enviar estos valores al emulador con el desplegable de la parte inferior izquierda (Speed 1X – 5X) y por último pulsamos el botón de “Play” de la izquierda.



Hecho esto, ya deberíamos empezar a ver en nuestra aplicación cómo se reciben periódicamente los valores de ubicación de nuestro listado de prueba.

Mapas en Android

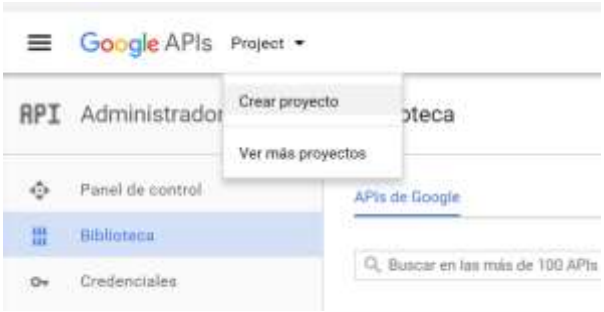
La API de Google Maps se integró con los *Google Play Services* allá por finales de 2012. Este cambio trajo consigo importantes mejoras, como la utilización de mapas vectoriales y mejoras en el sistema de caché, lo que proporcionaba mayor rendimiento, mayor velocidad de carga, y menor consumo de datos.

También llegó con un cambio en la forma en que los desarrolladores interactuarían con los mapas, pasando de los antiguos `MapActivity` y `MapView` a un nuevo tipo de *fragment* llamado `MapFragment`, con las ventajas que conlleva el uso de este tipo de componentes.

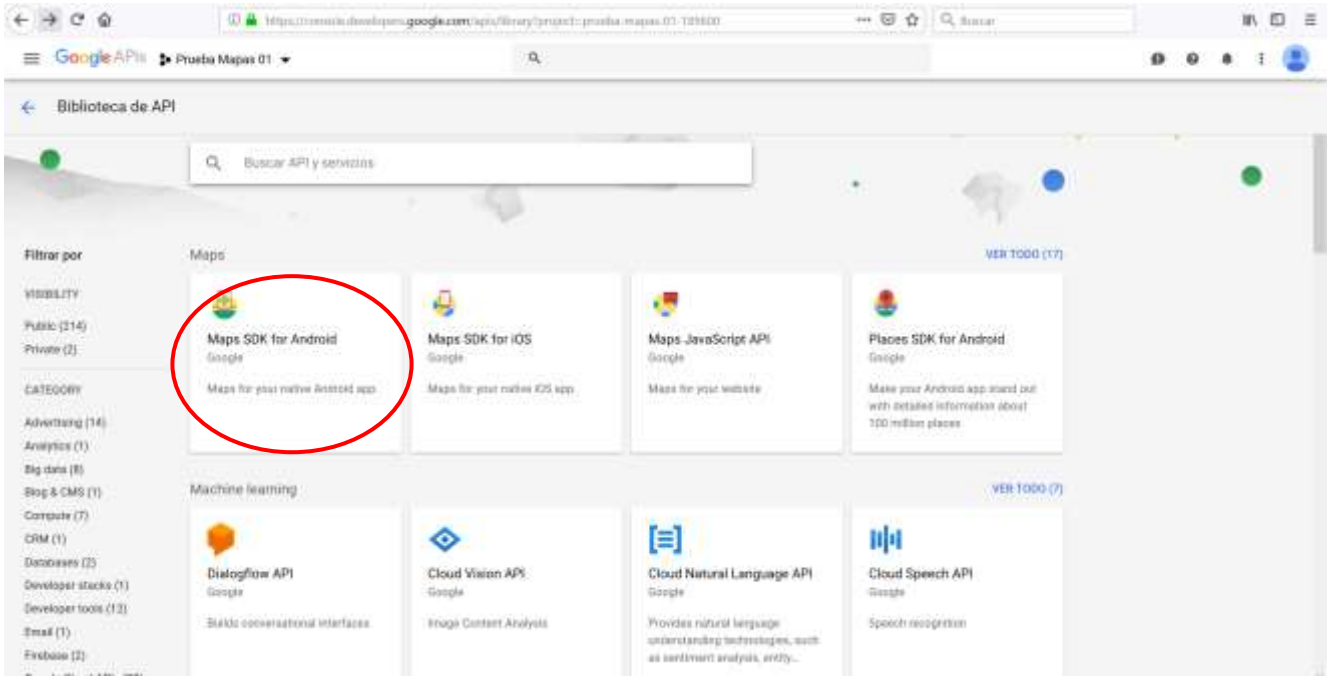
Antes de empezar a utilizar esta API en nuestras aplicaciones será necesario realizar algunos preparativos, y es que para hacer uso de los servicios de Google Maps es necesario que previamente generemos una *Clave de API* (o *API key*) asociada a nuestra aplicación. Éste es un proceso sencillo y se realiza accediendo a la Consola de Desarrolladores de Google.

Una vez hemos accedido, tendremos que crear un nuevo proyecto desde la lista desplegable que aparece en la parte superior derecha y seleccionando la opción “Crear proyecto…”.

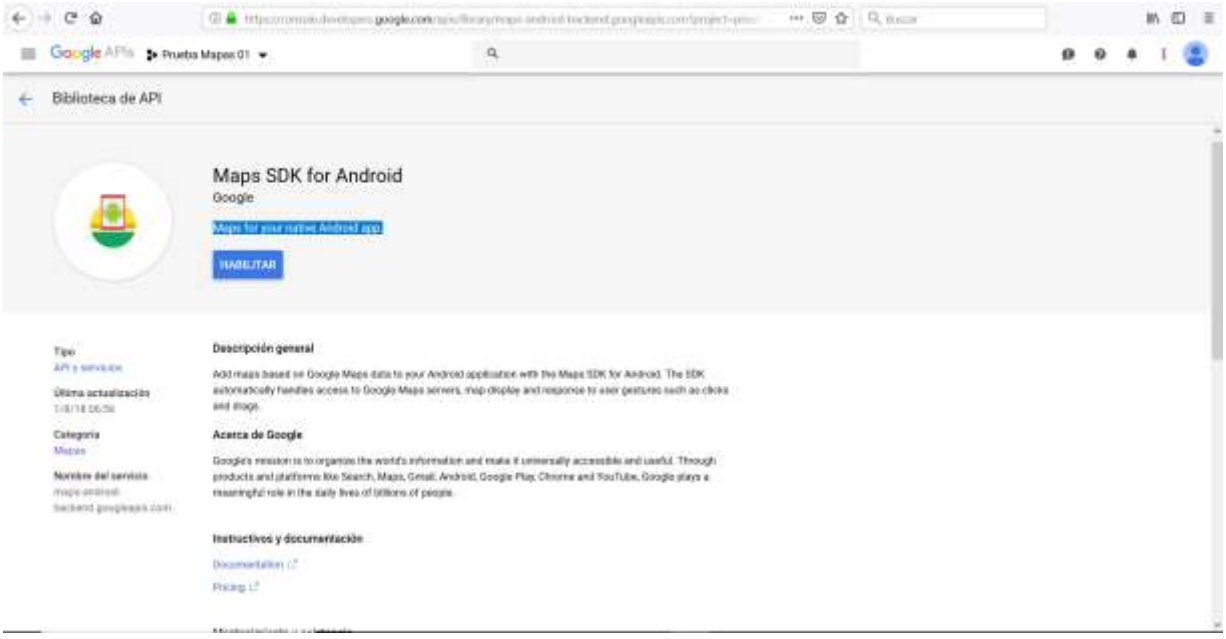
Aparecerá entonces una ventana que nos solicitará el nombre del proyecto. Introducimos algún nombre descriptivo, se generará automáticamente un ID único (que podemos editar aunque no es necesario), y aceptamos pulsando “Crear”.



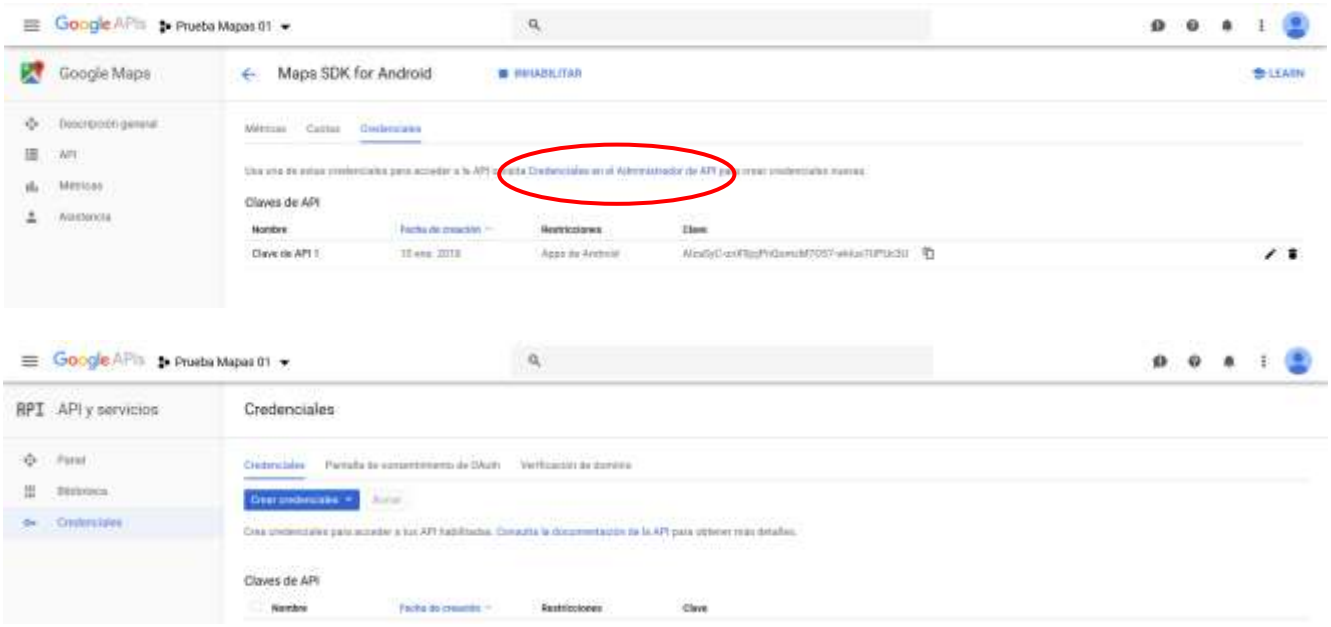
Una vez creado el proyecto llegamos a una página donde se nos permite seleccionar las APIs de Google que vamos a utilizar (como podéis ver la lista es bastante extensa). En nuestro caso particular vamos a seleccionar “*Maps SDK for Android*“.



Aparecerá entonces una ventana informativa con una breve descripción de la API y una advertencia indicando que se necesitará una clave de API para poder utilizarla. Por el momento vamos a activar la API haciendo click sobre la opción "*HABILITAR*" que aparece en la parte superior.

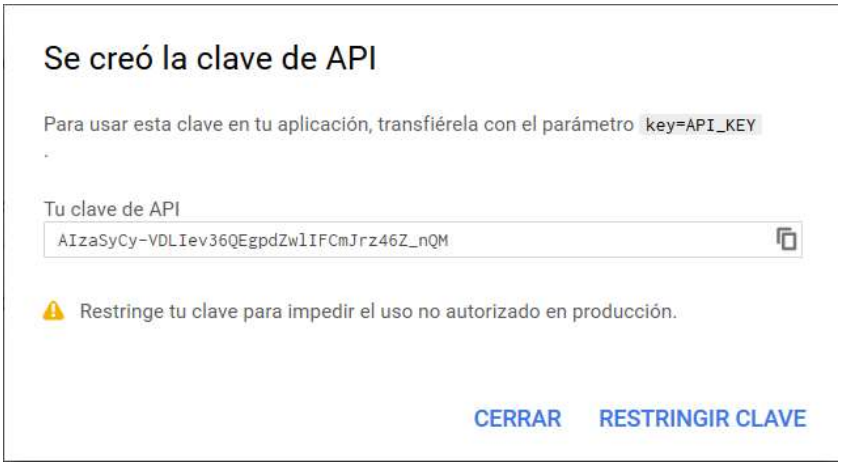


Una vez activada nos vuelve a aparecer la advertencia sobre la necesidad de obtener credenciales para el uso de la API por lo que, ahora sí, tendremos que iniciar el proceso de obtención de la clave. Pulsaremos para ello sobre el botón y/o el enlace "*Credenciales en el Administrador de API* "



Esto iniciará un pequeño asistente. En el primer paso, en el desplegable, seleccionamos *Clave de API* (Identifica tu proyecto con una clave de API simple para verificar la cuota y el acceso).

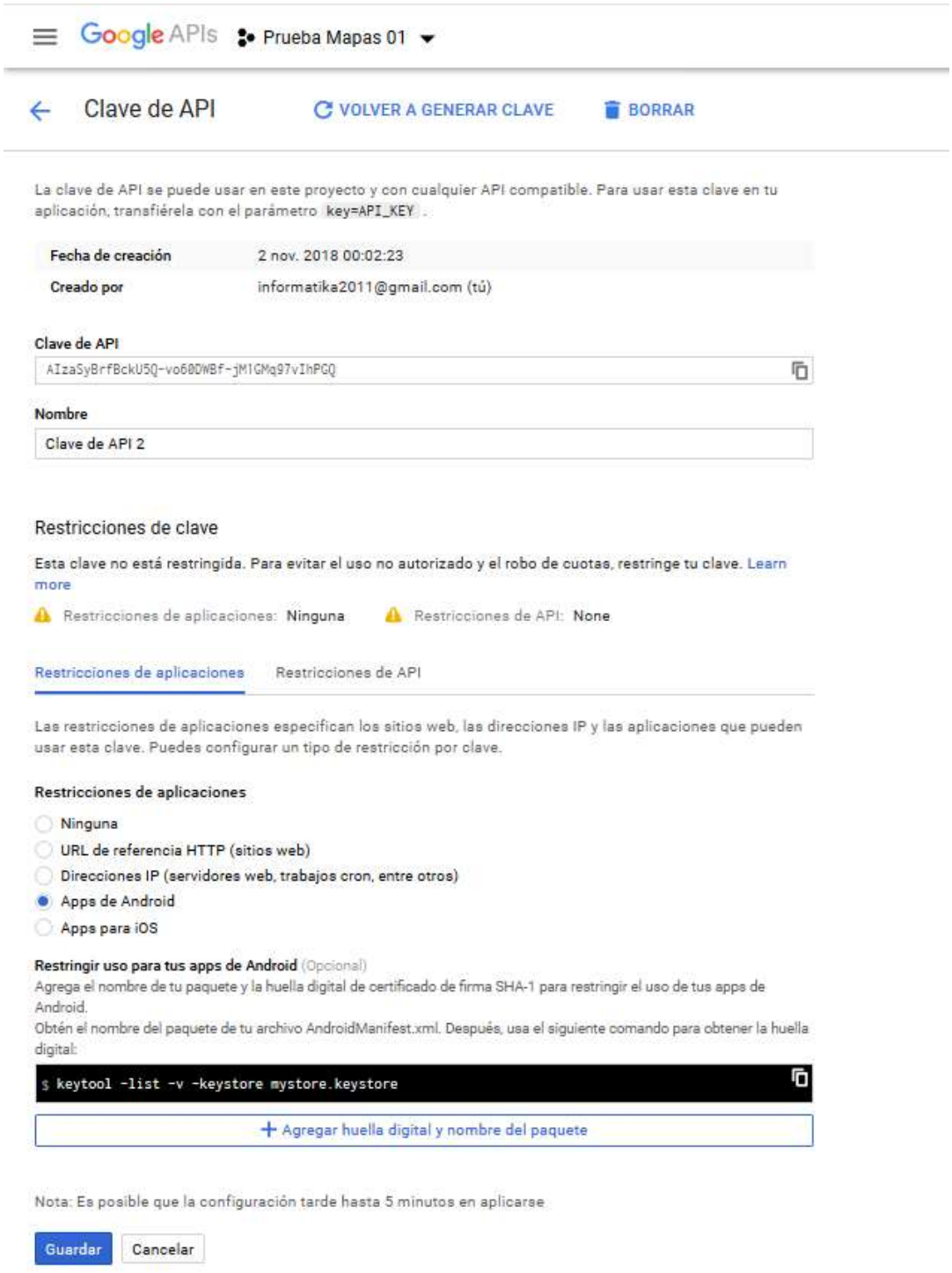
Tras esto, obtendremos una clave de API

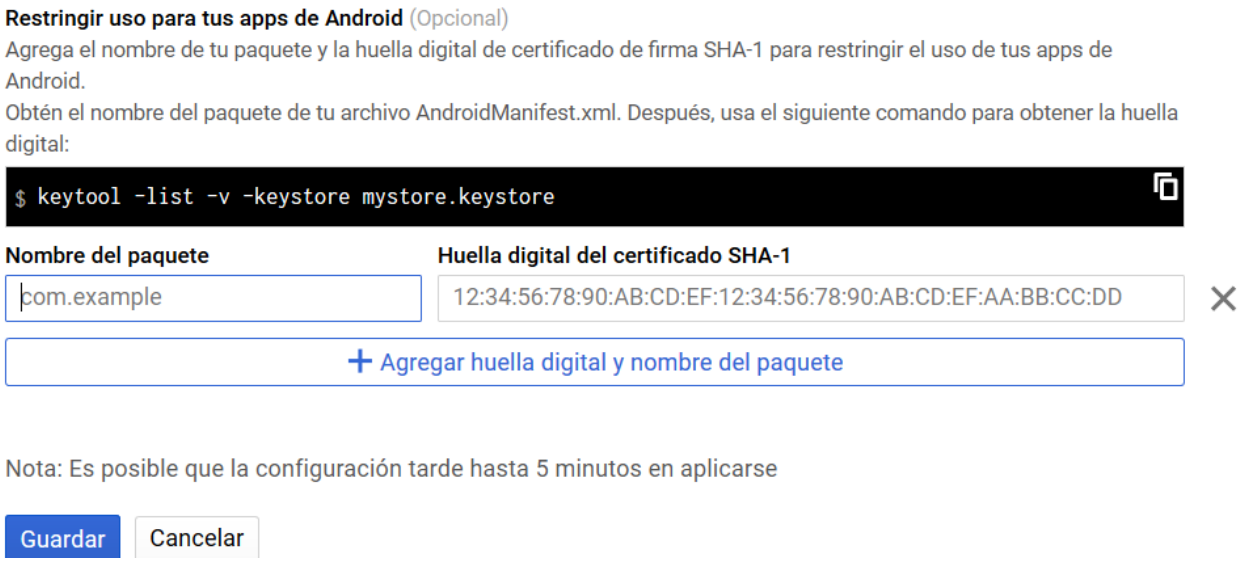


Hacer click sobre "*RESTRINGIR CLAVE*"

En el segundo paso tendremos que poner un nombre descriptivo a la **clave de API** que se va a generar, no es demasiado relevante, por lo que podemos dejar el que nos proponen por defecto. También en este paso se nos da la posibilidad de poder restringir el uso de este proyecto a determinadas aplicaciones concretas. Como es una práctica bastante recomendable vamos a ver como hacerlo. En nuestro caso, "Restricciones de aplicaciones" elegimos "*Apps de Android*". Pulsamos sobre el botón "+ *Agregar huella digital y nombre del paquete*" y veremos que se solicitan dos datos:

- Nombre de paquete
- Huella digital de certificado SHA-1





El primero de ellos es simplemente el paquete java principal que utilizaremos en nuestra aplicación. Lo indicaremos al crear nuestro proyecto en Android Studio (o si ya tenemos una aplicación creada podemos encontrarlo en el fichero *AndroidManifest.xml*, en el atributo package del elemento principal). En mi caso de ejemplo utilizaré el paquete "**`com.example.amaia.ejmapas`**".

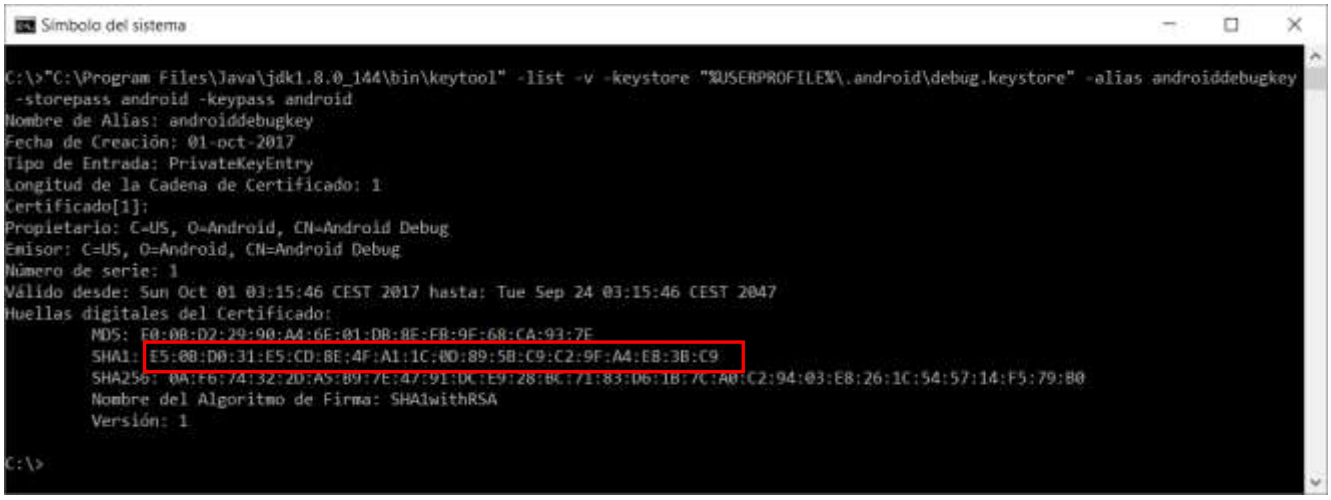
El segundo de los datos requiere más explicación. Toda aplicación Android debe ir firmada para poder ejecutarse en un dispositivo, tanto físico como emulado. Este proceso de firma es uno de los pasos que tenemos que hacer siempre antes de distribuir públicamente una aplicación. Adicionalmente, durante el desarrollo de la misma, para realizar las pruebas y la depuración del código, aunque no seamos conscientes de ello también estamos firmado la aplicación con un "*certificado de pruebas*". Pues bien, para obtener la huella digital del certificado con el que estamos firmando la aplicación podemos utilizar la utilidad *keytool* que se proporciona en el SDK de Java.

***Importante**: en este ejemplo vamos a obtener el SHA1 del certificado de pruebas, que es el que se utilizará para probar en el emulador, pero en caso de que nuestra aplicación se firmara de nuevo para subirla a la tienda de Google tendríamos que obtener de nuevo el SHA1 del nuevo certificado, o de lo contrario los mapas no se visualizarán.*

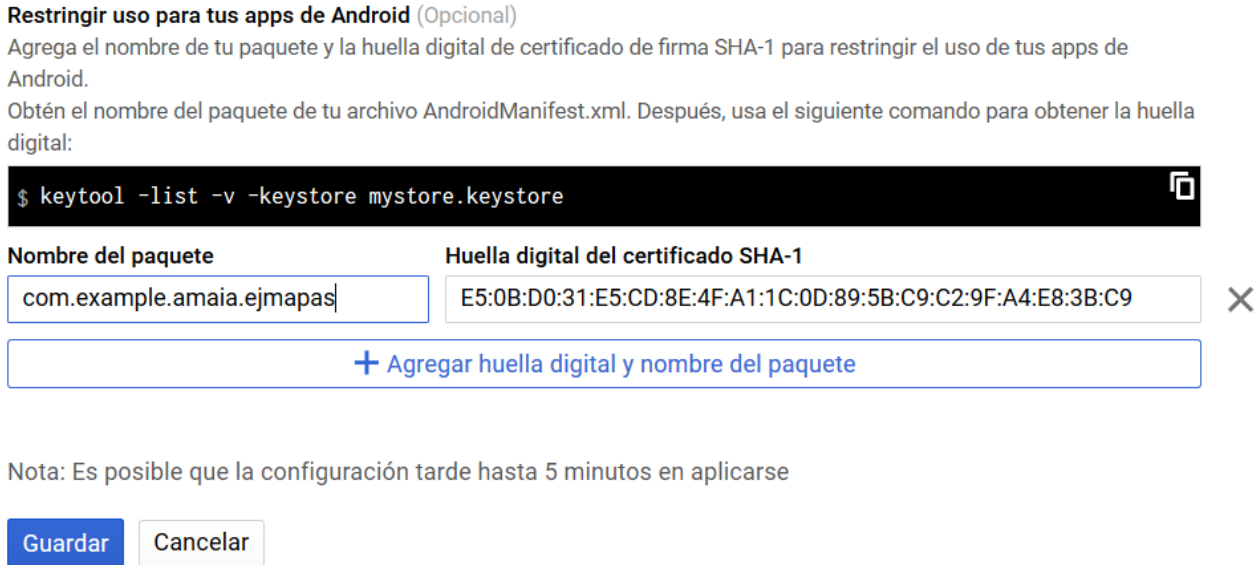
El certificado de pruebas debería encontrarse (en el caso de Windows) en la ruta "*<carpeta-usuario-logueado>\.android\debug.keystore*" (en mi caso: *C:\Users\Amaia\.android\debug.keystore*). Utilizaremos por tanto el siguiente comando para **obtener nuestra huella digital SHA-1**:

```
C:\>"C:\Program Files\Java\jdk1.8.0_144\bin\keytool" -list -v -
keystore "%USERPROFILE%\.android\debug.keystore" -alias
androiddebugkey -storepass android -keypass android
```

Suponiendo que tu instalación de Java está en la ruta "*C:\Program Files\Java\jdk1.8.0_144*". Si no es así sólo debes sustituir ésta por la correcta.



Una vez tenemos ya los dos datos necesarios (paquete y SHA-1) los introducimos en el asistente de obtención de credenciales y pulsamos el botón "*Guardar*".



Hecho esto, obtendremos por fin nuestra **clave de API**. La copiamos en lugar seguro (más tarde nos hará falta para nuestra aplicación Android).

Con esto ya habríamos concluido los preparativos iniciales necesarios para utilizar el servicio de mapas de Android en nuestras aplicaciones, por lo que vayamos a crear un proyecto de ejemplo en Android Studio.

Abrimos Android Studio y creamos un nuevo proyecto de Android, en mi caso lo he llamado "*EJMapas*", utilizando la plantilla "*Empty Activity*" y dejando todas las demás opciones por defecto. *Recuerda utilizar para el proyecto el mismo* ***paquete java*** *que hemos indicado durante la obtención de la* ***clave de API***.

Creado el proyecto, lo primero que haremos será abrir el fichero *build.gradle* de nuestro módulo principal para añadir la referencia a la librería específica de mapas de *Google Play Services*.

```
...
dependencies {
    . . .
    implementation 'com.google.android.gms:play-services-maps:16.1.0'
}
```

A continuación modificaremos el fichero *AndroidManifest.xml* para añadir tres elementos nuevos. En primer lugar, dentro del elemento `<application>` tendremos que añadir un nuevo elemento `<meta-data>` en el que se especificará la versión de los *Google Play Services* utilizada para compilar la aplicación.

```
<meta-data
    android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />
```

Inmediatamente después añadiremos otro nuevo `<meta-data>` donde indicaremos la *clave de API para Google Maps* que hemos obtenido a través de la consola de desarrolladores.

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="AIzaSyBrfBckU5Q-vo60DWBf-jM1GMq97vIhPGQ"/>
```

Por último, y fuera del elemento `<application>` añadiremos un nuevo elemento `<uses-feature>` para indicar que nuestra aplicación, o más concretamente Google Maps, hará uso de OpenGL ES versión 2.

```
<uses-feature
    android:glEsVersion="0x00020000"
    android:required="true"/>
```

Y con esto terminamos de configurar todo lo necesario. Ya podemos empezar a escribir nuestro código. Ahora nos vamos a limitar a mostrar un mapa en la pantalla principal de la aplicación. Y posteriormente veremos cómo añadir otras opciones o elementos al mapa.

Para esto tendremos simplemente que añadir el control correspondiente al layout de nuestra actividad principal. Lo añadiremos en forma de fragment de tipo `com.google.android.gms.maps.MapFragment`, quedando de la siguiente forma (no preocuparos si veis algún error de renderizado en el editor visual de layouts de Android Studio):

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/map"
    android:name="com.google.android.gms.maps.MapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Con esto ya podríamos ejecutar nuestra aplicación y deberíamos ver sin problemas un mapa a pantalla completa en su posición y zoom por defecto. Pero antes vamos a avanzar un poco más,

además de visualizar el mapa vamos a obtener una referencia a él desde nuestro código, referencia a partir de la cual podremos más adelante modificar sus propiedades y realizar determinadas acciones sobre el mapa.

Para esto, vamos a crear en primer lugar un atributo privado de tipo `GoogleMap` en nuestra actividad principal (que debe heredar de `AppCompatActivity`), y llamaremos `mapa`. Haremos además que nuestra actividad implemente la interfaz `OnMapReadyCallback`, en breve veremos para qué.

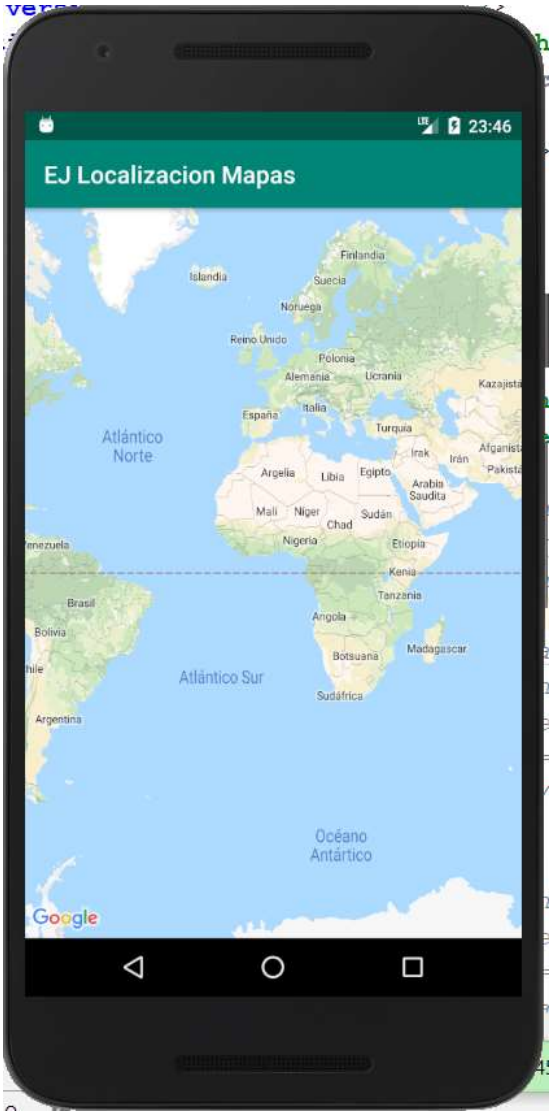
En el método `onCreate()` de la actividad obtendremos en primer lugar una referencia al `MapFragment` que hemos añadido a nuestro layout a través del fragment manager, y seguidamente llamaremos a su método `getMapAsync()` pasándole un objeto que implemente la interfaz `OnMapReadyCallback`, en nuestro caso la propia actividad principal.

Con esto conseguimos que en cuanto esté disponible la referencia al mapa (de tipo `GoogleMap`) incluido dentro de nuestro `MapFragment` se llame automáticamente al método `onMapReady()` de la interfaz. Por el momento la implementación de este método va a ser muy sencilla, limitándonos a guardar el objeto `GoogleMap` recibido como parámetro en nuestro atributo `mapa`.

Todo lo anterior quedaría de la siguiente forma:

```
public class MainActivity extends AppCompatActivity
        implements OnMapReadyCallback {

    private GoogleMap mapa;

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

        MapFragment mapFragment = (MapFragment)getFragmentManager()
                .findFragmentById(R.id.map);
        mapFragment.getMapAsync(this);
    }

    @Override
    public void onMapReady(GoogleMap googleMap) {
        mapa = googleMap;

    }
}
```

Y ahora sí, ejecutemos la aplicación para ver el aspecto de nuestro mapa.



Sobre este mapa, aunque básico, ya podemos movernos y hacer zoom utilizando los gestos clásicos.

A continuación haremos un repaso de las opciones básicas de un mapa:

- Cambiar las propiedades generales del mapa, como por ejemplo el tipo de vista (satélite, terreno, …) y la activación/desactivación de determinados controles superpuestos.
- Movernos por el mapa de forma programática.
- Obtener los datos de la posición actual mostrada en el mapa.
- Responder a los eventos principales del mapa.

Continuando con la aplicación, vamos a añadir varios botones superiores para mostrar el funcionamiento de las acciones anteriores. El layout lo mantendremos muy sencillo, quedaría de la siguiente forma:

```
<?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_height="match_parent"
    android:layout_width="match_parent" >

    <Button android:id="@+id/btnOpciones"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/opciones" />

    <Button android:id="@+id/btnMover"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toEndOf="@id/btnOpciones"
        android:text="@string/mover" />

    <Button android:id="@+id/btnAnimar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toEndOf="@id/btnMover"
        android:text="@string/animar" />

    <Button android:id="@+id/btnPosicion"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toEndOf="@id/btnAnimar"
        android:text="@string/pos" />

    <fragment
        android:id="@+id/map"
        android:name="com.google.android.gms.maps.MapFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_below="@id/btnOpciones"
        tools:layout="@layout/activity_main" />
</RelativeLayout>
```

Comentar que en la antigua versión de la API de Google Maps era bastante poco homogéneo el acceso y modificación de determinados datos del mapa. Por ejemplo, la consulta de la posición actual o la configuración del tipo de mapa se hacían directamente sobre el control `MapView`, mientras que la manipulación de la posición y el zoom se hacían a través del controlador asociado al mapa (`MapController`). Además, el tratamiento de las coordenadas y las unidades utilizadas eran algo peculiares, teniendo que estar continuamente convirtiendo de grados a microgrados y de éstos a objetos `GeoPoint`, etc.

Con la API actual, todas las operaciones se realizarán directamente sobre el objeto `GoogleMap`, el componente base de la API. Anteriormente, más arriba, hemos visto cómo obtener, dentro del evento `onMapReady()`, una referencia al objeto `GoogleMap` incluido en el `MapFragment` que añadimos a nuestro layout principal. Ahora, continuaremos a partir de ese punto para realizar diferentes acciones sobre el mapa.

Así, por ejemplo, para modificar el tipo de vista mostrada en el mapa podremos utilizar una llamada a su método `setMapType()`, pasando como parámetro el tipo de mapa, a elegir entre los siguientes:

- `GoogleMap.MAP_TYPE_NORMAL`. Mapa de carreteras normal.
- `GoogleMap.MAP_TYPE_SATELLITE`. Imágenes de satélite.
- `GoogleMap.MAP_TYPE_HYBRID`. Una mezcla de los dos anteriores.
- `GoogleMap.MAP_TYPE_TERRAIN`. Mapa topográfico.

Otra de las opciones que podemos cambiar son los indicadores y controles a mostrar sobre el mapa. Por ejemplo, para que se visualicen los controles de zoom, podemos llamar al método `getUiSettings()` del mapa para acceder a sus opciones de visualización, y sobre éste al método `setZoomControlsEnabled()`.

Como demostración añadiremos al primero de los botones de nuestra interfaz el cambio de las dos opciones comentadas, que quedaría de la siguiente forma:

```
public class MapasActivity extends AppCompatActivity
        implements OnMapReadyCallback {

    private GoogleMap mapa;
    private Button btnOpciones;

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

       ...

        btnOpciones = (Button)findViewById(R.id.btnOpciones);
        btnOpciones.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                //Cambiamos entre el mapa normal y el mapa satelite
                cambiarOpciones();
            }
        });

       ...

    //Cambiamos entre el mapa normal y el mapa satelite
    private void cambiarOpciones(){
        if (mapa.getMapType()==GoogleMap.MAP_TYPE_NORMAL) {
            mapa.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
            btnOpciones.setText("NORMAL");
        }
        else {
            mapa.setMapType(GoogleMap.MAP_TYPE_NORMAL);
            btnOpciones.setText("SATELITE");
        }
```

```
            mapa.getUiSettings().setZoomControlsEnabled(true);
        }

    }
```

El efecto sobre el mapa en la aplicación sería el siguiente:

En cuanto al movimiento sobre el mapa, podremos mover libremente nuestro punto de vista (o cámara, como lo han llamado los de Android) por un espacio 3D. De esta forma, ya no sólo podremos hablar de latitud-longitud (target) y zoom, sino también de orientación (bearing) y ángulo de visión (tilt). La manipulación de los 2 últimos parámetros unida a posibilidad actual de ver edificios en 3D de muchas ciudades nos abren un mundo de posibilidades.

El movimiento de la cámara se va a realizar siempre mediante la construcción de un objeto `CameraUpdate` con los parámetros necesarios. Para los movimientos más básicos como la actualización de la latitud y longitud o el nivel de zoom podremos utilizar la clase `CameraUpdateFactory` y sus métodos estáticos que nos facilitará un poco el trabajo.

Así por ejemplo, para cambiar sólo el nivel de zoom podremos utilizar los siguientes métodos para crear nuestro `CameraUpdate`:

- `CameraUpdateFactory.zoomIn()`. Aumenta en 1 el nivel de zoom.
- `CameraUpdateFactory.zoomOut()`. Disminuye en 1 el nivel de zoom.
- `CameraUpdateFactory.zoomTo(nivel_de_zoom)`. Establece el nivel de zoom.

Por su parte, para actualizar sólo la latitud-longitud de la cámara podremos utilizar:

- `CameraUpdateFactory.newLatLng(lat, long)`. Establece la latitud y longitud (lat-lng) expresadas en grados.

Si queremos modificar los dos parámetros anteriores de forma conjunta, también tendremos disponible el método siguiente:

- `CameraUpdateFactory.newLatLngZoom(lat, long, zoom)`. Establece la latitud, la longitud (lat-long) y el zoom.

Para movernos lateralmente por el mapa (*panning*) podríamos utilizar los métodos de scroll:

- `CameraUpdateFactory.scrollBy(scrollHorizontal, scrollVertical)`. Scroll expresado en píxeles.

Tras construir el objeto `CameraUpdate` con los parámetros de posición tendremos que llamar a los métodos `moveCamera()` o `animateCamera()` de nuestro objeto `GoogleMap`, dependiendo de si queremos que la actualización de la vista se muestre directamente en un solo paso o bien de forma animada.

Teniendo esto en cuenta, vamos por ejemplo, a centrar la vista en Madrid (España) con un zoom de nivel 5, y lo haremos en el segundo de los botones de la aplicación de ejemplo, quedando de la siguiente manera:

```
public class MapasActivity extends AppCompatActivity
        implements OnMapReadyCallback {

    private GoogleMap mapa;
    private Button btnOpciones;
    private Button btnMover;

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

        ...

        btnMover = (Button) findViewById(R.id.btnMover);
        btnMover.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                moverMadrid();
            }
        });
        ...

    private void moverMadrid (){
        CameraUpdate camUpd1 =
             CameraUpdateFactory.newLatLngZoom(new LatLng(40.41, -3.69),5);
        mapa.moveCamera(camUpd1);
    }

}
```

Veamos el resultado sobre la aplicación:

Además de los movimientos básicos que hemos comentado, si queremos modificar los demás parámetros de la cámara (orientación e inclinación) o varios de ellos simultáneamente tendremos disponible el método más general `CameraUpdateFactory.newCameraPosition()` que recibe como parámetro un objeto de tipo `CameraPosition`. Este objeto lo construiremos indicando todos los parámetros de la posición y orientación de la cámara a través de su método `Builder()`.

Así, por ejemplo, si quisiéramos mostrar de una forma más estética el monumento a Alfonso XII de Madrid podríamos hacerlo de la siguiente forma, lo haremos en el tercer botón de demostración de la aplicación de ejemplo:

```
public class MapasActivity extends AppCompatActivity
        implements OnMapReadyCallback {

    private GoogleMap mapa;
    private Button btnOpciones;
    private Button btnMover;
    private Button btnAnimar;

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

        btnAnimar = (Button) findViewById(R.id.btnAnimar);
        btnAnimar.setOnClickListener(new View.OnClickListener() {
```

```
                @Override
                public void onClick(View view) {
                    animarMadrid();
                }
            });
        }

            ...

        private void animarMadrid () {
            LatLng madrid = new LatLng(40.417325, -3.683081);

            CameraPosition camPos = new CameraPosition.Builder()
                    .target(madrid)
                    .zoom(19)
                    .bearing(45)
                    .tilt(70)
                    .build();

            CameraUpdate camUpd3 = CameraUpdateFactory.newCameraPosition(camPos);

            mapa.animateCamera(camUpd3);
        }
    }
```

Como podemos comprobar, mediante este mecanismo podemos modificar todos los parámetros de posición de la cámara (o sólo algunos de ellos) al mismo tiempo. En nuestro caso de ejemplo hemos centrado la vista del mapa sobre el parque de El Retiro de Madrid (España), con un nivel de zoom de 19, una orientación de 45 grados para que el noreste esté hacia arriba y un ángulo de visión de 70 grados de forma que veamos en 3D el monumento a Alfonso XII en la vista de mapa NORMAL. En la siguiente imagen vemos el resultado:

Como se puede apreciar, la API actual de mapas facilita bastante el posicionamiento dentro del mapa, y el uso de las clases `CameraUpdate` y `CameraPosition` resulta bastante intuitivo.

Bien, hemos visto cómo modificar nuestro punto de vista sobre el mapa, pero si el usuario se mueve de forma manual por él, ¿cómo podemos conocer en un momento dado la posición de la cámara?

Pues igual de fácil, mediante el método `getCameraPosition()`, que nos devuelve un objeto `CameraPosition` como el que ya conocíamos. Accediendo a los distintos métodos y propiedades de este objeto podemos conocer con exactitud la posición de la cámara, la orientación y el nivel de zoom. En la aplicación de ejemplo añadiremos la demostración de esto en el cuarto de los botones añadidos, que mostrará un mensaje *Toast* con la posición actual de la cámara.

```
public class MapasActivity extends AppCompatActivity
        implements OnMapReadyCallback {

    private GoogleMap mapa;
    private Button btnOpciones;
    private Button btnMover;
    private Button btnAnimar;
    private Button btnPosicon;

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

        ...

        btnPosicon = (Button) findViewById(R.id.btnPosicion);
        btnPosicon.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                obtenerPosicion();
            }
        });
    }

        ...

    private void obtenerPosicion () {
        CameraPosition cameraPosicion = mapa.getCameraPosition();

        LatLng coordenadas = cameraPosicion.target;
        double latitud = coordenadas.latitude;
        double longitud = coordenadas.longitude;

        Toast.makeText(this, "Lat: "+latitud  +'\n' +
              "Long: "+ longitud, Toast.LENGTH_SHORT).show();

    }
}
```

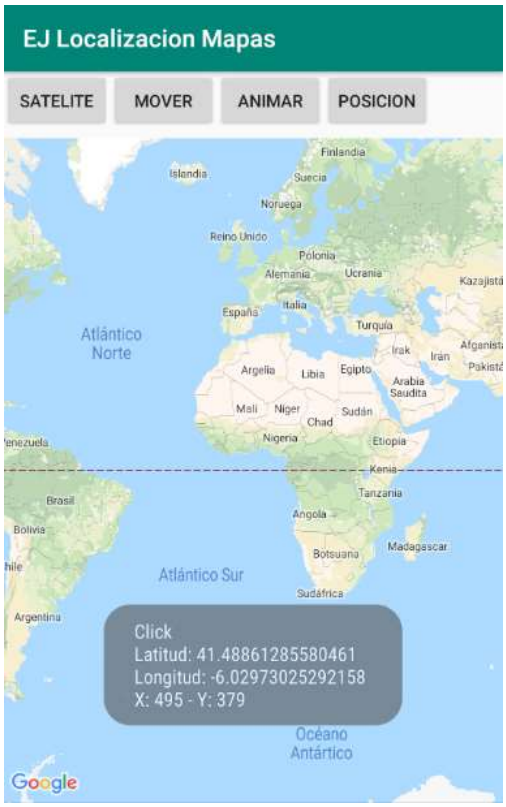
Veamos un ejemplo sobre la propia aplicación:

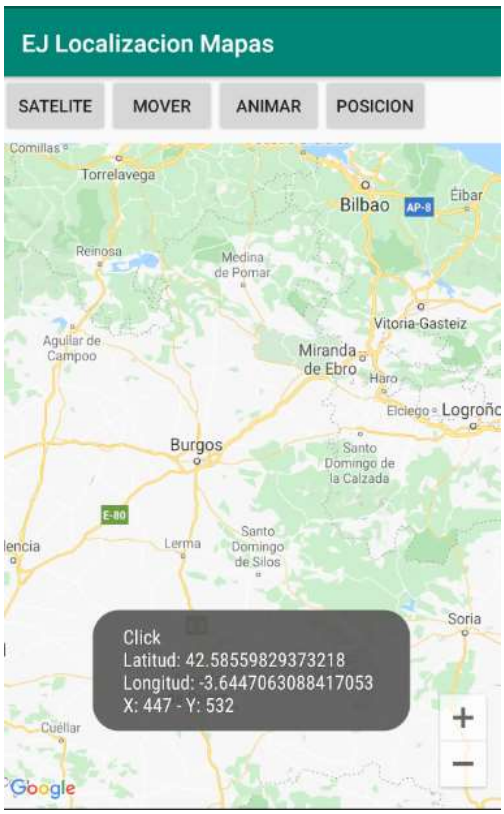
Por último, vamos a ver cómo responder a ciertos eventos del mapa, entre los que destacamos el *click* y *el cambio de posición*. Para responder a estos eventos definiremos los *listener* correspondientes sobre nuestro objeto `GoogleMap`.

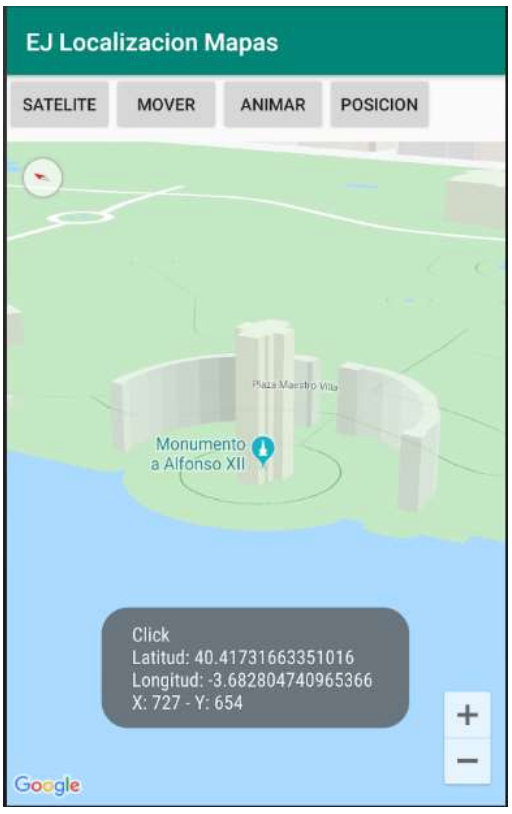
Para responder a los clicks del usuario sobre el mapa definiremos el evento `onMapClick()` llamando al método `setOnMapClickListener()` del mapa. Dentro de éste recibiremos un objeto `LatLng` con la posición geográfica, la que mostraremos en un mensaje tipo *Toast* junto a la posición X-Y de pantalla (dentro de la vista actual del mapa) donde el usuario ha pulsado. Para obtener esta posición usaremos el método `toScreenLocation()` del objeto `Projection` que podemos obtener a partir del mapa llamando a `getProjection()`.

```
@Override
public void onMapReady(GoogleMap googleMap) {
    mapa = googleMap;

    mapa.setOnMapClickListener(new GoogleMap.OnMapClickListener() {
        @Override
        public void onMapClick(LatLng latLng) {
            Projection proj = mapa.getProjection();
            Point coord = proj.toScreenLocation(latLng);

            Toast.makeText(MainActivity.this,"Click " +
                    "\nLatitud: "+ latLng.latitude+
                    "\nLongitud: "+ latLng.longitude+
                    "\nX: "+ coord.x + " - Y: "+ coord.y,
                    Toast.LENGTH_SHORT).show();
        }
    });
}
```

Un ejemplo sobre la aplicación sería el siguiente:







Lo siguiente es centrarnos en la creación y gestión de marcadores, y en el dibujo de elementos gráficos, como líneas y polígonos, sobre el mapa.

Añadiremos tres nuevos botones, en la parte de debajo, para probar estas otras nuevas funcionalidades.

Empezamos por la creación de marcadores. Rara es la aplicación Android que hace uso de mapas sin utilizar también este tipo de elementos para resaltar determinados puntos de interés sobre el mapa. Agregar un marcador básico a un mapa resulta tan sencillo como llamar al método `addMarker()` pasándole la posición, en forma de objeto `LatLng`, y el texto a mostrar en la ventana de información del marcador. En nuestra aplicación de ejemplo añadiremos al primer botón una acción de este tipo, de forma que cuando lo pulsemos se añada automáticamente un marcador sobre España con el texto “Pais: España“.

```
public class MapasActivity extends AppCompatActivity
        implements OnMapReadyCallback {

    private GoogleMap mapa;
    private Button btOpciones;
    private Button btMover;
    private Button btAnimar;
    private Button btPosicion;

    private Button btMarcador;

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

        ...

        btMarcador = (Button)findViewById(R.id.btnMarcador);
        btMarcador.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                insertarMarcador();
            }
        });

    }

    ...

    private void insertarMarcador() {
        marker = mapa.addMarker(new MarkerOptions()
                .position(new LatLng(40.3936945, -3.701519))
                .title("Pais: España"))
                .snippet("La capital de España es Madrid"));

    marker.showInfoWindow();
    //marker.hideInfoWindow();
    }

    ...

}
```

Así de sencillo, basta con llamar al método `addMarker()` pasando como parámetro un nuevo objeto `MarkerOptions` sobre el que establecemos la posición del marcador (método `position()`) y el texto a incluir en la ventana de información del marcador (métodos `title()` para el título y `snippet()` para el resto del texto). Si ejecutamos la aplicación de ejemplo y pulsamos el botón “MARCADORES” aparecerá el siguiente marcador sobre el mapa:

Si pulsamos sobre el marcador, la vista se centrará en la posición del marcador y se mostrará la ventana de información con el texto indicado (ver imagen superior).

Pero vemos que además aparecen por defecto en la esquina inferior derecha dos botones para mostrar dicha ubicación en la aplicación de Google Maps y para calcular la ruta al lugar marcado. Si no queremos que aparezcan estos botones, debemos llamar previamente al método `setMapToolbarEnabled(false)` sobre nuestro objeto `GoogleMap`. Podríamos hacer esto por ejemplo dentro del método `onMapReady()` tras obtener la referencia al mapa:

```
public void onMapReady(GoogleMap googleMap) {
    mapa = googleMap;

    mapa.getUiSettings().setMapToolbarEnabled(false);

    ...

}
```

Si no nos gusta el comportamiento por defecto al pulsar sobre un marcador, también tenemos la posibilidad de definirlo de forma personalizada asignando al mapa dicho evento como cualquiera de los comentados anteriormente. En este caso el listener se asignará al mapa mediante el método `setOnMarkerClickListener()` y sobrescribiremos el método `onMarkerClick()`. Dicho método

recibe como parámetro el objeto `Marker` pulsado, de forma que podamos identificarlo accediendo a su información (posición, título, texto, …). Veamos un ejemplo donde mostramos un *toast* con el título del marcador pulsado:

```
@Override
public void onMapReady(GoogleMap googleMap) {
    mapa = googleMap;

    mapa.getUiSettings().setMapToolbarEnabled(false);

    ...
    //Al pulsar sobre el marcador
    mapa.setOnMarkerClickListener(new GoogleMap.OnMarkerClickListener() {
        public boolean onMarkerClick(Marker marker) {
            Toast.makeText(
                    MainActivity.this,
                    "Marcador pulsado:\n" +
                            marker.getTitle(),
                    Toast.LENGTH_SHORT).show();

            return true;
        }
    });
}
```

Con el código anterior, si pulsamos sobre el marcador de España aparecerá el siguiente mensaje informativo:

Salvo esto último, todo lo visto sobre marcadores corresponde al comportamiento por defecto de la API, sin embargo también es posible por supuesto personalizar determinadas cosas, como por ejemplo el aspecto de los marcadores (tan sólo decir que mediante los métodos `icon()` y

`anchor()` del objeto `MakerOptions` que hemos visto antes es posible utilizar una imagen personalizada para mostrar como marcador en el mapa. En la documentación oficial (en inglés) podéis encontrar un ejemplo de cómo hacer esto).

Para finalizar, vamos a ver cómo dibujar líneas y polígonos sobre el mapa, elementos muy comúnmente utilizados para trazar rutas o delimitar zonas del mapa. Para realizar esto vamos a actuar una vez más directamente sobre la vista de mapa, sin necesidad de añadir *overlays* o similares (para quienes recuerden versiones anteriores de la API), y ayudándonos de los objetos `PolylineOptions` y `PolygonOptions` respectivamente.

Para dibujar una linea lo primero que tendremos que hacer será crear un nuevo objeto `PolylineOptions` sobre el que añadiremos, utilizando su método `add()`, las coordenadas (latitud-longitud) de todos los puntos que conformen la línea. Tras esto estableceremos el grosor y color de la línea llamando a los métodos `width()` y `color()` respectivamente, y por último añadiremos la línea al mapa mediante su método `addPolyline()` pasándole el objeto `PolylineOptions` recién creado.

En nuestra aplicación de ejemplo haremos esto en el segundo botón de demostración para dibujar un rectángulo sobre España. Veamos cómo queda el código:

```
public class MapasActivity extends AppCompatActivity
        implements OnMapReadyCallback {

    private GoogleMap mapa;
    ...
    private Button btMarcador;
    private Button btLineas;

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

        ...

        btLineas = (Button)findViewById(R.id.btnLineas);
        btLineas.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                mostrarLineas();
            }
        });
    }

    ...

    private void mostrarLineas()
    {
        //Dibujo con Lineas
        PolylineOptions lineas = new PolylineOptions()
                .add(new LatLng(45.0, -12.0))
                .add(new LatLng(45.0, 5.0))
                .add(new LatLng(34.5, 5.0))
                .add(new LatLng(34.5, -12.0))
                .add(new LatLng(45.0, -12.0));
```

```
            lineas.width(8);
            lineas.color(Color.RED);

            mapa.addPolyline(lineas);
        }
        ...

}
```

Ejecutando esta acción en el emulador veríamos lo siguiente:

Pues bien, esto mismo podríamos haberlo logrado mediante el dibujo de polígonos, cuyo funcionamiento es muy similar. Para ello crearíamos un nuevo objeto `PolygonOptions` y añadiremos las coordenadas de sus puntos en el sentido de las agujas del reloj. En este caso no es necesario cerrar el circuito (es decir, que la primera coordenada y la última sean iguales) ya que se hace de forma automática. Otra diferencia es que para polígonos el ancho y color de la linea los estableceríamos mediante los métodos `strokeWidth()` y `strokeColor()`. Además, el dibujo final del polígono sobre el mapa lo haríamos mediante `addPolygon()`. En nuestro caso de ejemplo, implementado en el tercer botón de demostración, quedaría como sigue:

```
public class MapasActivity extends AppCompatActivity
        implements OnMapReadyCallback {

    private GoogleMap mapa;
    ...

    private Button btMarcador;
    private Button btLineas;
    private Button btPoligono;

    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ...

        btPoligono = (Button)findViewById(R.id.btnPoligono);
        btPoligono.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                mostrarPoligono();
            }
        });
    }

    ...

    private void mostrarPoligono()
    {
        //Dibujo con Poligonos
        PolygonOptions rectangulo = new PolygonOptions()
                .add(new LatLng(45.0, -12.0),
                        new LatLng(45.0, 5.0),
                        new LatLng(34.5, 5.0),
                        new LatLng(34.5, -12.0));

        rectangulo.strokeWidth(8);
        rectangulo.strokeColor(Color.GREEN);

        mapa.addPolygon(rectangulo);
    }

    ...

}
```

El resultado al ejecutar la acción en el emulador debería ser exactamente igual que el anterior, pero en este caso el recuadro será verde.

Actividad

Realizar una aplicación en la que situemos en el mapa Museos, Campos de futbol, Estaciones de esquí, Montes, … lo que más os apetezca, leyendo las posiciones desde un fichero de texto y/o Bases de Datos en el que previamente hayas guardado el nombre del lugar y la posición.