Desarrollo Avanzado de Software - Curso 2023 / 2024

Laboratorio 08.- Uso de APIs de Google (I)

Contenido:

Objetivos: Aprender el funcionamiento básico de algunas de las APIs que ofrece Google.

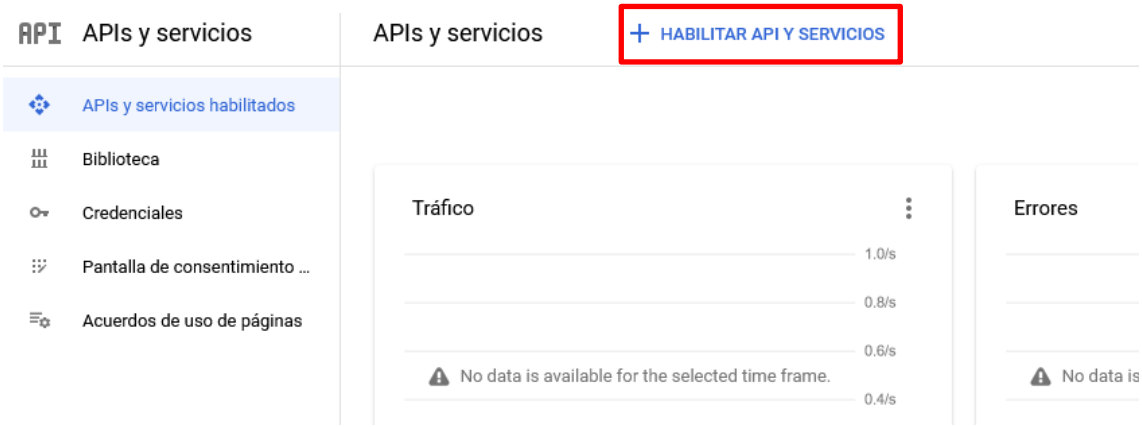
1. Google Maps

1.1. Habilitar Maps SDK para Android

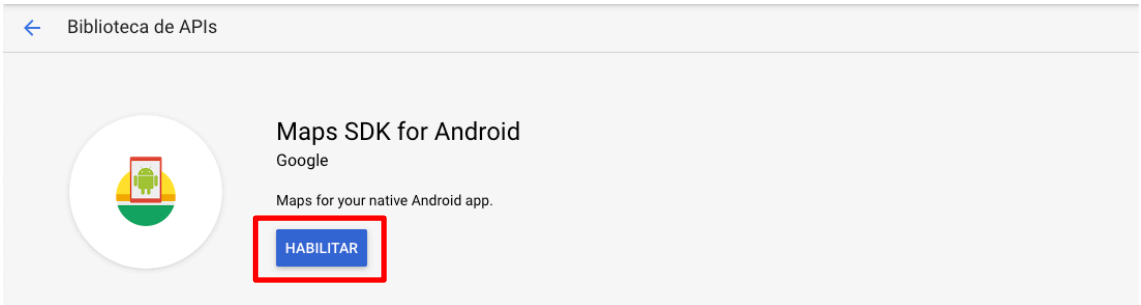
Para poder acceder a los servicios del API de Google Maps necesitamos una clave que permita identificar nuestra aplicación. Esta clave nos la proporciona Google mediante la consola de desarrolladores (http://console.developers.google.com).

Hay que crear un nuevo proyecto

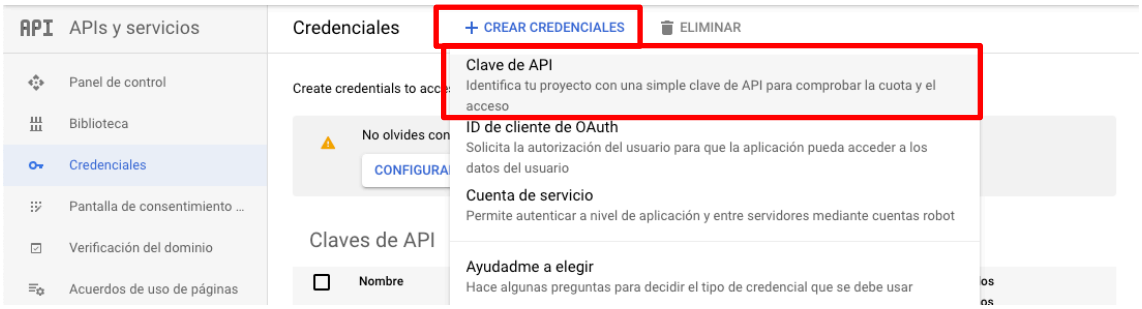
Una vez creado el proyecto, hay que habilitar las APIs que se deseen



Entre todas las APIs disponibles, hay que seleccionar "Maps SDK forAndroid" y habilitarla

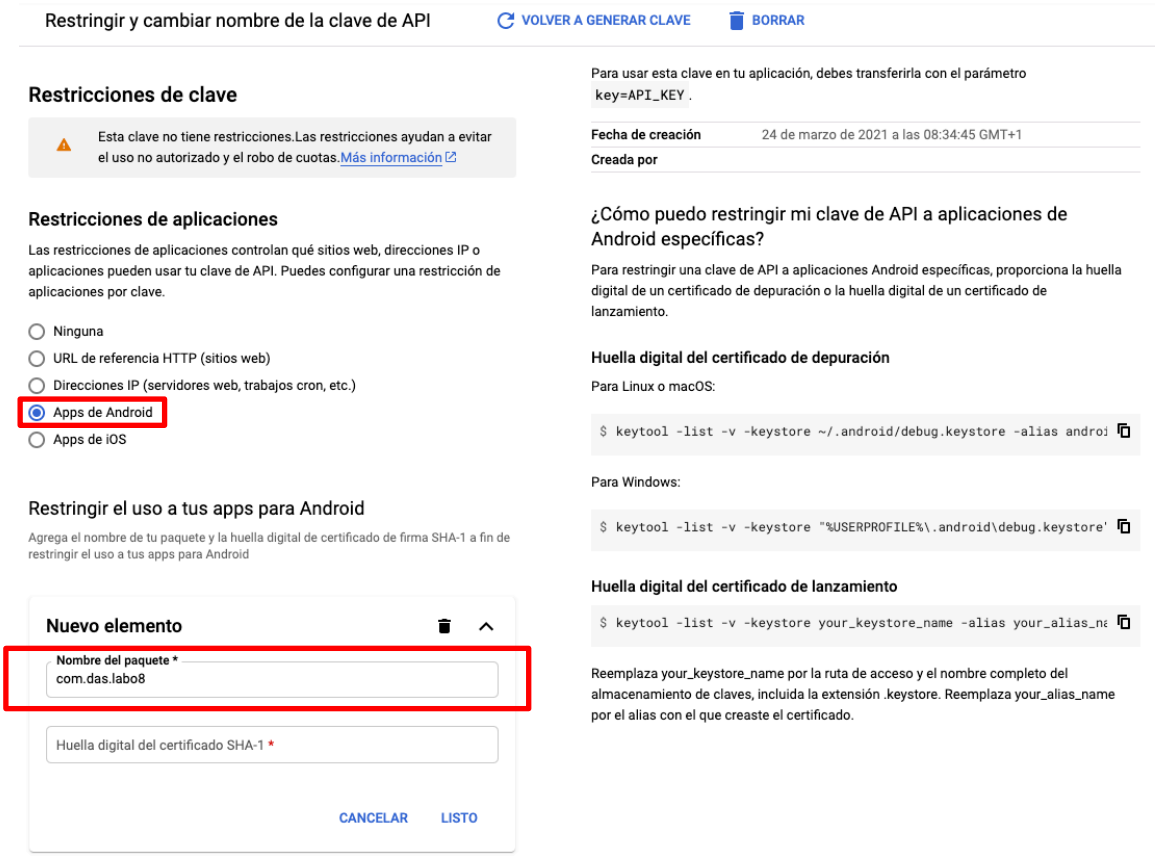


Una vez habilitada, en el apartado de Credenciales, hay que generar una nueva Clave de API pulsando en Crear Credenciales



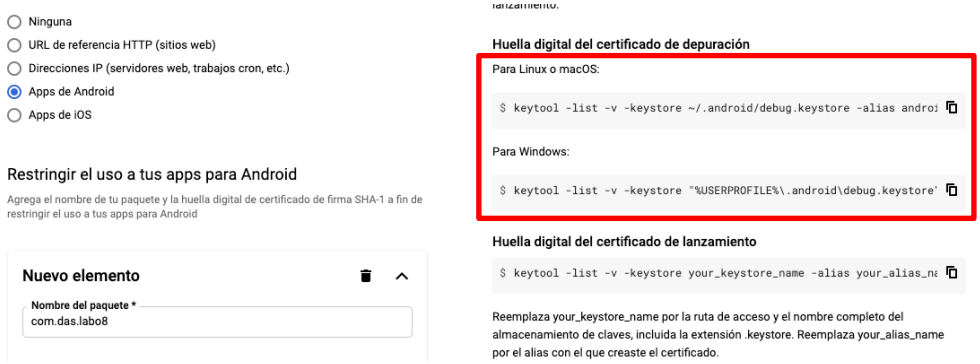
1.2. Restricciones de la clave de la API

Muchas APIs tienen cuotas gratuitas, pero luego son de pago, así que es conveniente restringir su uso. En este caso la restringiremos para sólo funcione en nuestra aplicación concreta. Elegiremos que sólo funcione en "Aplicaciones Android", introduciremos el nombre del paquete de nuestra aplicación y nos pedirá la huella digital de nuestro certificado SHA-1. Para obtenerla, hay que seguir los siguientes pasos.

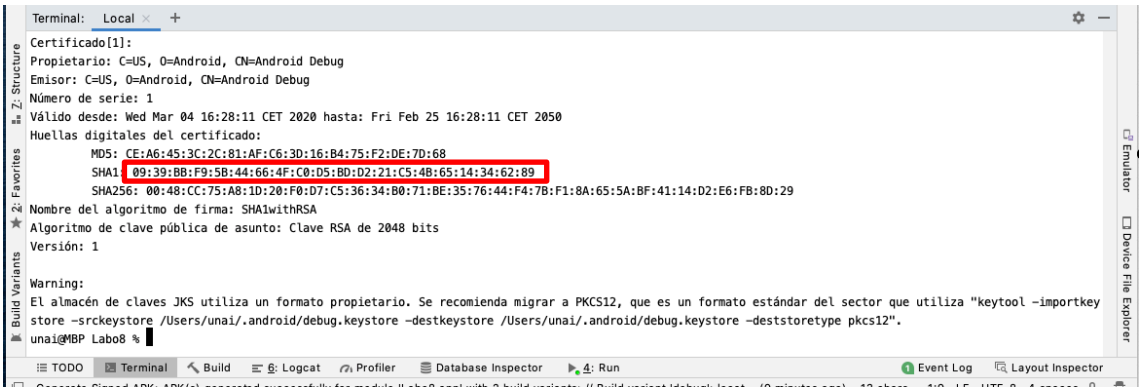


Cuando ejecutas o depuras tu proyecto desde el IDE, Android Studio automáticamente firma tu app con un certificado de depuración generado por las herramientas del SDK de Android. La primera vez que ejecutas o depuras tu proyecto en Android Studio, el IDE crea automáticamente el almacén de claves y el certificado de depuración en `$HOME/.android/debug.keystore`, y configura el almacén de claves y las contraseñas de claves.

Utilizando el almacén de claves por defecto del IDE (debug.keystore), copiamos el comando que nos indican en la consola para obtener la huella digital del certificado de depuración

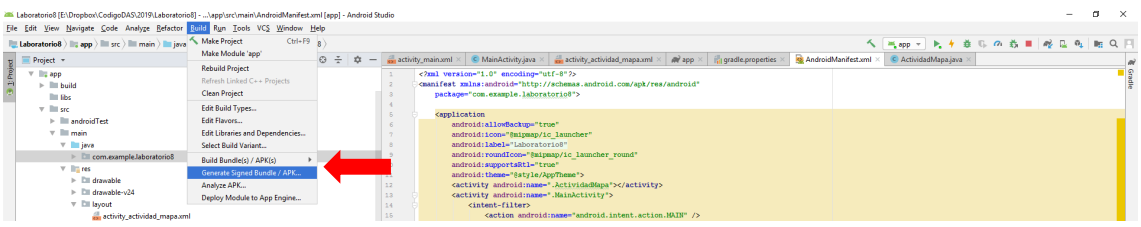


Ejecutamos el comando `keytool` en el directorio donde tengamos instalado el JDK que utiliza Android Studio (`C:\Program Files\Android\Android Studio\jbr\bin`) desde la terminal de Android Studio (si tiene permisos de administrador) o desde consola y entre los datos que nos mostrará, está la huella digital SHA-1. La copiamos en la consola de Google Developers y finalizamos el proceso de creación de la clave API. Para saber donde está el directorio del JDK abrimos el diálogo de Settings en Android Studio. Vamos a "Build, Execution, Deployment › Build Tools › Gradle".



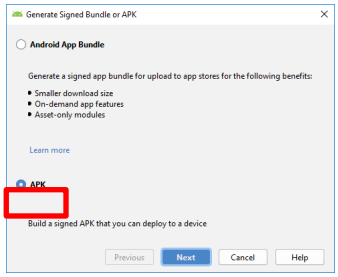
1.3. Firmar la app con un nuevo almacén de claves

Si se quisiera firmar para lanzarla en Google Play, por ejemplo, sería necesario firmar la aplicación con un almacen de claves nuevo ya que no se podría utilizar el debug.keystore generado por Android Studio. Para ello, A través del menú de Android Studio *Build → Generate Signed Bundle/APK*

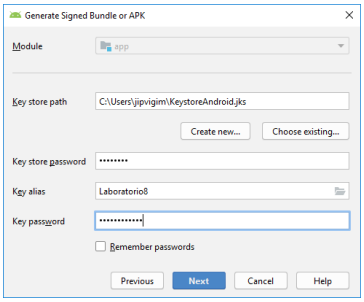


Elegiremos que queremos generar un APK firmado

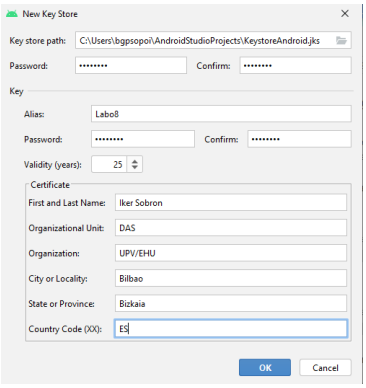
Desarrollo Avanzado de Software - Curso 2023 / 2024



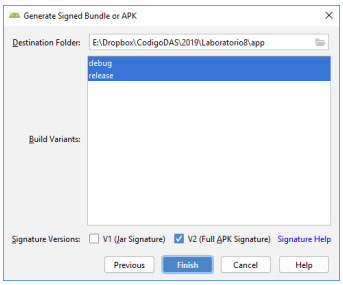
Si es la primera vez que firmamos un APK, tendremos que crear un almacén de certificados (Key store) y un certificado nuevo (Key alias). Si ya tenemos un almacén y un certificado creados, podremos reutilizarlos.



Si tenemos que crear un certificado nos pedirá los datos correspondientes.



Se elige la carpeta donde se desea dejar el APK firmado, se elige que se quiere utilizar tanto para debug como para release y que el tipo de firma es V2.



1.4. Uso de la API en la aplicación

En el manifiesto de nuestra aplicación, dentro del elemento ‹application›, añadimos la información de la clave que acabamos de crear:

```
<application
    ...
    <meta-data
        android:name="com.google.android.geo.API_KEY"
        android:value="LA CLAVE INDICADA EN LA CONSOLA" />
</application>
```

La inclusión de la clave directamente en el manifiesto se considera poco seguro y se sugiere agregarla en el archivo `local.properties` que se encuentra en el directorio raíz del proyecto. Para ello, abre el archivo `build.gradle` a nivel del proyecto y agrega el siguiente código al elemento `dependencies` en `buildscript`.

```
classpath "com.google.android.libraries.mapsplatform.secrets-gradle-plugin:secrets-gradle-plugin:2.0.1"
```

A continuación, abre el archivo `build.gradle` a nivel del módulo y agrega el siguiente código al elemento `plugins`.

```
id 'com.google.android.libraries.mapsplatform.secrets-gradle-plugin'
```

Guarda el archivo y sincroniza tu proyecto con Gradle.

Abre el archivo `local.properties` en el directorio de nivel de proyecto y, luego, agrega el siguiente código. Reemplaza YOUR_API_KEY por tu clave de API.

```
MAPS_API_KEY=YOUR_API_KEY
```

Guarda el archivo.

En tu archivo AndroidManifest.xml, ve a com.google.android.geo.API_KEY y actualiza el android:value attribute de la siguiente manera:

```
<meta-data

android:name="com.google.android.geo.API_KEY"
android:value="$(MAPS_API_KEY)"/>
```

También hay que incluir entre las dependencias de la aplicación (en el fichero build.gradle del módulo) la dependencia de la API

```
implementation 'com.google.android.gms:play-services-maps:18.1.0'
```

Para trabajar con los mapas de Google Maps se usan Fragments (ver https://developers.google.com/maps/documentation/android-sdk/map?hl=es-419). Para añadir un mapa a la interfaz hay que definir un fragmento en el layout e indicar que su clase es: *com.google.android.gms.maps.SupportMapFragment.*

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:id="@+id/fragmentoMapa"
```

```
android:layout_width="match_parent"
android:layout_height="match_parent"/>
```

La actividad, deberá extender a *FragmentActivity* e implementar el método `onMapReadyCallback`

```
public class ActividadMapa extends FragmentActivity implements OnMapReadyCallback {
...
}
```

Para poder trabajar con el mapa deberemos utilizar el identificador que le hayamos asignado al Fragment donde se encuentra el mapa y llamar al método `getMapAsync(…)`.

```
SupportMapFragment elfragmento =

(SupportMapFragment)getSupportFragmentManager().findFragmentById(R.id.fragmentoMapa);
elfragmento.getMapAsync(this);
```

Este método lo que hará será llamar al método `onMapReady`, que recibe como parámetro el objeto *GoogleMap* con el que ya podremos trabajar.

```
void onMapReady(GoogleMap googleMap) {

    elmapa = googleMap;
        …
```

Una vez que tenemos un objeto **`elmapa`** de la clase *GoogleMap* se puede trabajar con él y con objetos de tipo *CameraUpdate*, *CameraUpdateFactory* y *LatLng* para ajustar la visualización del mapa a lo que deseemos. Por ejemplo, la operación `setMapType` de la clase *GoogleMap* nos permite definir el tipo de visualización que queremos tener del mapa:

```
elmapa.setMapType(GoogleMap.MAP_TYPE_NORMAL);
```

Sus posibles valores son:

- MAP_TYPE_NORMAL
- MAP_TYPE_HYBRID
- MAP_TYPE_SATELLITE
- MAP_TYPE_TERRAIN

Para el posicionamiento de la cámara se usa un objeto de tipo *CameraUpdate* y para movimientos básicos, como la actualización de la latitud y longitud o el nivel de zoom, se pueden usar los métodos de la clase *CameraUpdateFactory*.

Por ejemplo:

- `CameraUpdateFactory.zoomIn()` → Aumenta el zoom en 1.
- `CameraUpdateFactory.zoomOut()` → Disminuye en 1 el zoom.
- `CameraUpdateFactory.zoomTo(numero)` → Fija el nivel de zoom.
- `CameraUpdateFactory.newLatLng(LatLng)` → Fija las coordenadas (en grados).
- `CameraUpdateFactory.newLatLngZoom(LatLng, zoom)` → Fija las coordenadas (en grados) y el nivel de zoom
- `CameraUpdateFactory.scrollBy(scrollHorizontal, scrollVertical)` → Indica el desplazamiento (*panning*) expresado en píxeles.

Para obtener objetos de tipo *LatLng* se usa el constructor de la clase y se le proporciona como parámetros la latitud y la longitud.

```
LatLng nuevascoordenadas= new LatLng(43.26,-2.95);
```

Mediante las operaciones de la clase *CameraUpdateFactory*, construimos un objeto de tipo *CameraUpdate* que será el que se le proporcione al mapa para modificar su aspecto.

```
CameraUpdate actualizar = CameraUpdateFactory.newLatLngZoom(new LatLng(43.26, -2.95),9);
```

Si se usa el método `moveCamera()`, la nueva vista se mostrará directamente y si se usa el método `animateCamera()` se mostrará la transición de forma animada.

```
elmapa.moveCamera(actualizar);
```

Además de los movimientos básicos que hemos comentado, se pueden modificar los demás parámetros de la cámara o varios de ellos simultáneamente a través del método `newCameraPosition()` de la clase *CameraUpdateFactory*. El método `newCameraPosition` recibe como parámetro un objeto de tipo *CameraPosition*. Este objeto se construye indicando todos los parámetros de la posición de la cámara a través de su método `Builder()` y sus atributos.

- Target → Las coordenadas (mediante un objeto *LatLng*)
- Zoom → El nivel de zoom
- Bearing → La orientación de la cámara (en grados siendo el 0 en el norte)
- Tilt → El ángulo de la cámara (en grados siendo 0 nuestra vertical)

```
CameraPosition Poscam = new CameraPosition.Builder()
        .target(nuevascoordenadas)
        .zoom(6)
        .bearing(54)
        .tilt(5)
        .build();
CameraUpdate otravista = CameraUpdateFactory.newCameraPosition(Poscam);
elmapa.animateCamera(otravista);
```

También se puede acceder en todo momento a la configuración de la cámara a través del método `getCameraPosition()` del objeto *GoogleMap*. Esta operación nos devuelve un objeto de tipo *CameraPosition* que nos permite acceder a sus atributos de target, zoom, etc.

```
CameraPosition posicion= elmapa.getCameraPosition();
LatLng coord = posicion.target;
```

Uno de los usos más habituales de los mapas es indicar gráficamente la posición de ciertos elementos. Para ello se usan los marcadores. Para añadir un marcador:

```
elmapa.addMarker(new MarkerOptions()
        .position(new LatLng(43.25, -3.92))
        .title("El marcador"));
```

Para programar acciones cuando se pulsa un marcador se usa un listener sobre el mapa, donde se recibe como parámetro el marcador que se ha pulsado. Si se devuelve false, además del código que se haya programado, se ejecutará el comportamiento por defecto de los marcadores (mostrar la información y centrar el mapa). Si se devuelve true, sólo se ejecutará el código programado.

```
elmapa.setOnMarkerClickListener(new GoogleMap.OnMarkerClickListener() {
    public boolean onMarkerClick(Marker marker) {
        …
        return false;
    }
});
```

Para saber dónde hace "click" el usuario en el mapa, existe el listener sobre el mapa que devuelve como parámetro las coordenadas sobre las que el usuario ha pulsado.

```
elmapa.setOnMapClickListener(new GoogleMap.OnMapClickListener() {
    @Override
    public void onMapClick(LatLng latLng) {
        …
    }
});
```

2. Aplicación a realizar: Cálculo y dibujo de "rutas"

Desarrollaremos una aplicación que le muestre al usuario un mapa centrado en su posición actual y que le permita ir añadiendo marcadores indicando una "ruta" (el punto de origen de la ruta será su posición actual). Cada vez que añada un nuevo marcador se le indicará la distancia (en metros) al marcador anterior y la distancia total de la ruta.

Modificaremos la aplicación para que muestre al usuario gráficamente mediante una línea la ruta que está siguiendo al moverse, actualizando su posición en el mapa con cada cambio de localización.

💡 Para poder dibujar líneas sobre un mapa hay que utilizar la clase Polyline.