Programación Multimedia y Dispositivos Móviles

UD 15. Mapas

Javier Carrasco

Curso 2024 / 2025

Mapas

15. Mapas

En la actualidad, rara es la aplicación que no implementa un mapa para ubicar servicios de la propia aplicación o cualquier otro tipo de información mediante geoposicionamiento. Existen en el mercado diferentes APIs que pueden ayudar a realizar esta tarea en cuestión. En este capítulo se centrará la atención en dos de las más utilizadas, **Mapbox**[1] y la API de **Google Maps**[2].

Antes de comenzar, es necesario conocer como emular el **GPS** del dispositivo el emulador. En la ventana de este, deberás pulsar sobre el icono con tres puntos en la parte superior. En la pestaña *Location*, deberás utilizar el buscador para localizar el punto deseado y, por último, pulsar el botón **Set Location**.

Al buscar ubicaciones, aparecerá la opción para guardar el punto (*Save Point*) y poder así crear una lista de ubicaciones favoritas que podrás ir cambiando y ver así como actúa el mapa.

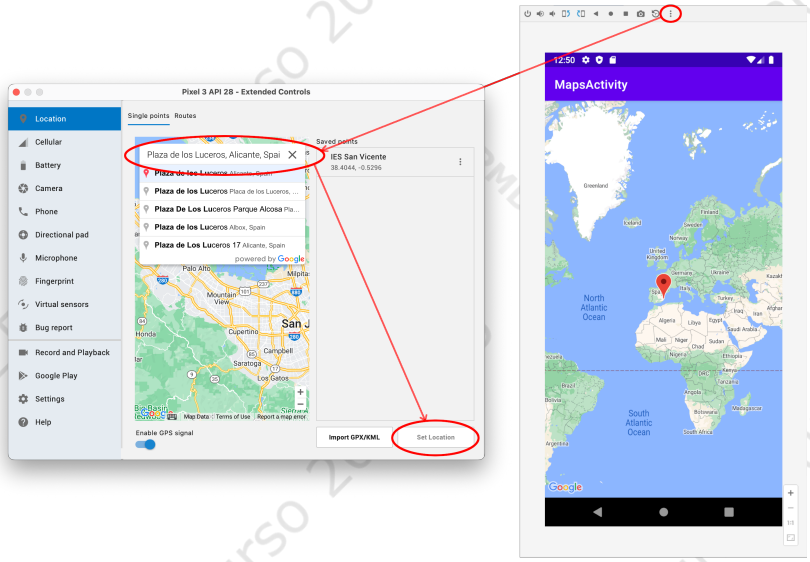

Figura 1

Si utilizas un dispositivo físico, estos pasos no serán necesarios, ya que utilizará la ubicación del dispositivo.

15.1. Mapbox

Mapbox es un generador, o proveedor, de mapas *online* que pueden ser utilizados tanto por páginas web como por aplicaciones móviles. Además, ofrece una serie de ventajas sobre otras APIs como:

1 Mapbox (https://www.mapbox.com/)
2 Google Maps (https://mapsplatform.google.com/)

- Personalizable 100%.
- *Open source*.
- Multiplataforma.
- Funcionamiento *in-app*, es decir, todo se ejecuta u ocurre en la aplicación.
- SDK muy similar a la de *Google Maps* para Android y *MapKit* para iOS.
- Gratis hasta los primeros 25.000 usuarios activos con el SDKs *for Mobile*.

Durante el desarrollo de este punto, se tratará con la versión disponible en el momento de la revisión, la **versión 10.16.2.** del **Maps SDK for Android** de **Mapbox**.[3]

15.1.1. Configuración

Para comenzar a utilizar **Mapbox**, el primer paso consistirá en registrarse en su página web. Este es un paso de lo más sencillo, no requiere más que indicar un nombre de usuario, un correo electrónico y una contraseña, el resto de campos son opcionales.

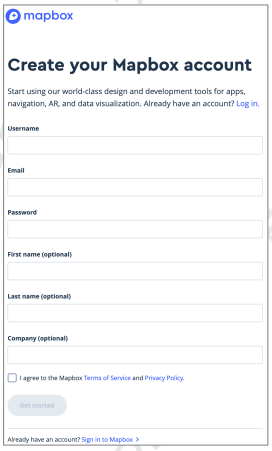

Figura 2

Una vez estés registrado, y hayas validado el correo electrónico para activar la cuenta, ya podrás acceder al panel de control que ofrece Mapbox para ver su configuración y comprobar el estado.

Destacar que, en la pantalla de inicio aparecerá creado un ***token público***, éste será el que se utilice para poder hacer funcionar Mapbox en la aplicación.

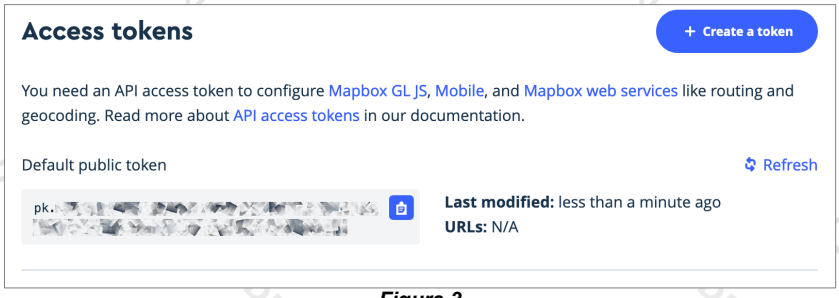

Figura 3

El *token público* siempre estará visible en el panel de control de Mapbox, ahora, deberá crearse un ***token privado*** desde la consola de Mapbox utilizando la opción ***Create a token***, deberás indicar un nombre y marcar al menos la opción DOWNLOADS:READ. Hecho esto, ya puedes pulsar el botón ***Create a token***. Una vez creado, asegúrate de copiar el *token* para después utilizarlo, verás un mensaje indicando que sólo tienes una oportunidad para copiarlo por motivos de seguridad.

3 *Maps SDK for Android* (https://docs.mapbox.com/android/maps/guides/)

A continuación, se creará un nuevo proyecto en **Android Studio** con API mínima superior o igual a la **26**, es importante conocer que la versión mínima para utilizar Mapbox es la 21 según versiones anteriores. EL proyecto partirá de una *Empty Views Activity* vacía como en capítulos anteriores. Una vez creado el proyecto *MyFirstMapbox*, deberán realizarse las siguientes acciones.

En primer lugar, si eres usuario de **GitHub**, no te interesa que se suban los *tokens* y queden a la vista. Una vez creado el proyecto, edita el fichero ***.gitignore*** creado en la raíz del proyecto y añade la siguiente línea para excluir el fichero `developer-config.xml` que se creará en ***values*** y que contendrá el *token* secreto.

```
<TU_APP>/app/src/main/res/values/developer-config.xml
```

A continuación, se añadirán los *tokens* al proyecto. Para el **token privado** se creará un fichero llamado `gradle.properties`, dentro del directorio `.gradle` del usuario, `<USER_HOME>/.gradle/` en el caso de Mac, añadiendo a este la siguiente línea.

```
MAPBOX_DOWNLOADS_TOKEN=YOUR_SECRET_MAPBOX_ACCESS_TOKEN
```

Sustituye el texto *YOUR_SECRET_MAPBOX_ACCESS_TOKEN* por tu *token* privado. Ahora, para añadir el *token* público, crea el fichero `developer-config.xml` en ***values*** y añade la siguiente línea al fichero. Si no vas ha hacer uso de GitHub, puedes añadirla en `strings.xml`.

```
<string name="mapbox_access_token">YOUR_MAPBOX_ACCESS_TOKEN</string>
```

Deberás sustituir el texto *YOUR_MAPBOX_ACCESS_TOKEN* por el *token* público que aparece en el panel de control de Mapbox indicado como *Default public token*. A continuación, se añadirá al *Manifest* los permisos de localización. Añade también el permiso para el uso de Internet, al fin y al cabo será necesario para descargar información.

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

El siguiente paso será añadir la dependencias[4] necesarias al fichero `build.gradle(Module: app)` y sincronizar.

```
implementation("com.mapbox.maps:android:11.0.0")
```

Por último, para terminar de añadir dependencias, en el fichero `settings.gradle.kts` de proyecto, deberá añadirse la referencia a la descarga del SDK de Mapbox, configurando el repositorio de *Maven* para añadir el usuario y contraseña, tras sincronizar el *Gradle*, ya estarán las dependencias satisfechas.

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()

```

4 Descarga directa de Mapbox (https://docs.mapbox.com/android/maps/guides/install/#add-the-dependency)

```
        maven {
            authentication {
                create<BasicAuthentication>("basic")
            }
            url = uri("https://api.mapbox.com/downloads/v2/releases/maven")
            credentials {
                // Do not change the username below.
                // This should always be `mapbox` (not your username).
                username = "mapbox"
                // Use the secret token you stored in gradle.properties as the password
                password = providers.gradleProperty("MAPBOX_DOWNLOADS_TOKEN").get()
            }
        }
    }
}
```

El siguiente paso será añadir el componente Mapbox a la *activity* principal, puedes utilizar el siguiente código y adaptarlo a tu aplicación. Puede ser el elemento raíz de la actividad.

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

    <com.mapbox.maps.MapView
        android:id="@+id/mapView"
        mapbox:mapbox_cameraTargetLat="40.7128"
        mapbox:mapbox_cameraTargetLng="-74.0060"
        mapbox:mapbox_cameraZoom="9.0"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</FrameLayout>
```

Ya en la `MainActivity.kt`, se puede añadir la siguiente línea en el método `onCreate()` de la clase principal de la aplicación.

```
binding.mapView.mapboxMap.loadStyle(Style.SATELLITE_STREETS)
```

Realmente no haría falta, ya que una vez configurado, y añadido el componente *MapView* al *layout*, ya lo verás funcionar si lanzas la aplicación. Esta línea simplemente establecerá el estilo de visualización del mapa.

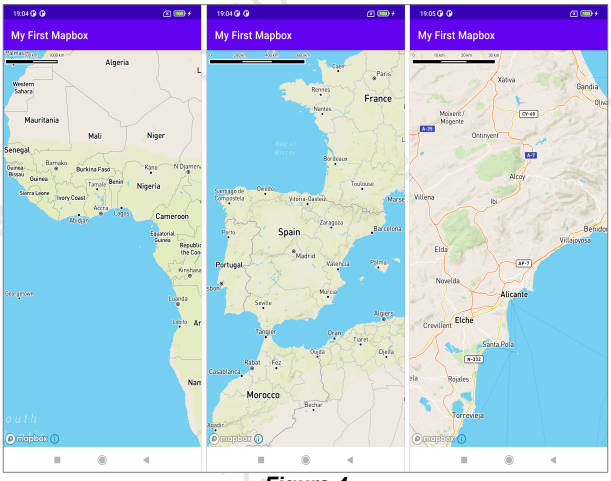

Figura 4

Para el correcto funcionamiento de Mapbox entre Android y OpenGL, éste define su propio ciclo de vida, por lo que deberán sobrescribirse todos los métodos del ciclo de vida de una *activity* de la siguiente forma.

```
override fun onStart() {
    super.onStart()
    binding.mapView.onStart().apply {
        Log.d("onStart", "Start Mapbox!!")
    }
}
```

En este caso se utiliza *apply*, ya que `mapView` siempre existirá, pero si no se estuviese seguro, se puede utilizar el método `let`, que se ejecutará cuando el objeto en cuestión **no sea nulo**. Los métodos del ciclo de vida de una *activity* que se deben sobrecargar son los siguientes, además de *onStart()*, para Mapbox.

```
override fun onStop() {
    super.onStop()
    binding.mapView.onStop().apply {
        Log.d("onStop", "Stop Mapbox!!")
    }
}

override fun onLowMemory() {
    super.onLowMemory()
    binding.mapView.onLowMemory().apply {
        Log.d("onLowMemory", "LowMemory Mapbox!")
    }
}

override fun onDestroy() {
    super.onDestroy()
```

```
    binding.mapView.onDestroy().apply {
        Log.d("onDestroy", "Destroy Mapbox!")
    }
}
```

Si revisas la documentación de Mapbox, verás que ya no es necesario implementar explícitamente estos métodos, ya que se aplicará automáticamente cuando la versión de *appcompact* sea 1.3.0. o superior, quedando totalmente integrado en *androidx.lifecycle*.

Otra opción es iniciar el mapa desde código, sin necesidad de poner en el componente *MapView* del *layout* las propiedades *mapbox*.

```
// Create a MapboxMap
val mapbox = binding.mapView.mapboxMap
mapbox.setCamera(
    CameraOptions.Builder()
        .center(Point.fromLngLat(-0.5295719, 38.4044311))
        .pitch(0.0) // inclinación, 0.0 es vertical
        .zoom(9.0)
        .bearing(0.0) // rotación, 0.0 es norte
        .build()
)
```

Ahora se verá como modificar algunas opciones del mapa, por ejemplo, desactivar el logo y la información de Mapbox, o hacer que la brújula únicamente aparezca cuando no se esté mirando al norte.

```
// Carga de un estilo desde al API Estilos de Mapbox.
mapbox.loadStyle(Style.OUTDOORS)

// Ya se puede añadir información o hacer otros ajustes.
binding.mapView.logo.enabled = false // Desactiva el logo.
binding.mapView.attribution.enabled = false // Desactiva la atribución.

// Mantiene la brújula siempre activada.
binding.mapView.compass.fadeWhenFacingNorth = false
```

En lo referente a los estilos que ofrece *Mapbox*[5], se dispone de los típicos que se pueden utilizar con `Style`; *DARK*, *LIGHT*, *MAPBOX_STREETS*, *OUTDOORS*, *SATELLITE*, *SATELLITE_STREETS*, *TRAFFIC_DAY* y *TRAFFIC_NIGHT*. Pero también pueden generarse mapas propios mediante **Mapbox Studio**[6], utilizando como fuente una URI.

```
mapbox.loadStyle("mapbox://styles/mapbox/dark-v11")
```

Otro aspecto que debe conocerse de Mapbox, es que forma los mapas por **capas** (*layers*), las cuales permiten realizar ciertas personalizaciones en tiempo de ejecución. Observa como se podrían obtener las capas de un mapa según el estilo aplicado.

5 Mapbox styles (https://docs.mapbox.com/api/maps/styles/#mapbox-styles)
6 Mapbox Studio (https://docs.mapbox.com/studio-manual/guides/)

```
mapbox.loadStyle(Style.OUTDOORS) { style ->
    // Se obtienen las capas del estilo.
    for (singleLayer in style.styleLayers) {
        // Se muestra el nombre de cada capa.
        Log.d("singleLayer", singleLayer.id)
    }
}
```

El bucle *for* generará las siguientes líneas en *Logcat*, en el que podrá verse todas y cada una de las capas que forman el mapa según el estilo aplicado.

```
2023-...45 singleLayer            es.javiercarrasco.myfirstmapbox      D  land
2023-...45 singleLayer            es.javiercarrasco.myfirstmapbox      D  landcover
2023-...45 singleLayer            es.javiercarrasco.myfirstmapbox      D  national-park
2023-...45 singleLayer            es.javiercarrasco.myfirstmapbox      D  national-park_tint-band
2023-...45 singleLayer            es.javiercarrasco.myfirstmapbox      D  landuse
2023-...45 singleLayer            es.javiercarrasco.myfirstmapbox      D  pitch-outline
...
2023-...45 singleLayer            es.javiercarrasco.myfirstmapbox      D  water
```

Conociendo el nombre de las capa, se puede modificar según interese, o aplicar cambios según las acciones que se hagan sobre el mapa. Supón que quieres que el agua del mapa sea de color rojo. Basta con añadir las siguientes líneas en la misma carga del estilo.

```
mapbox.loadStyle(Style.OUTDOORS) { style ->
    // Agua de color rojo.
    val singleLayer = style.getLayerAs<FillLayer>("water")
    singleLayer!!.fillColor(Color.RED)
}
```

15.1.2. Localización y permisos

Antes de seguir, para simplificar la configuración del mapa, la configuración inicial vista anteriormente se añadirá al método `onMapReady()`.

```
private fun onMapReady() {
    // Create a MapboxMap
    val mapbox = binding.mapView.mapboxMap

    mapbox.setCamera(
        CameraOptions.Builder()
            .center(Point.fromLngLat(-0.5295719, 38.4044311))
            .pitch(0.0) // inclinación, 0.0 es vertical
            .zoom(12.0)
            .bearing(0.0) // rotación, 0.0 es norte
            .build()
    )

    // Ya se puede añadir información o hacer otros ajustes.
    binding.mapView.logo.enabled = true // Desactiva el logo.
```

```
    binding.mapView.attribution.enabled = false // Desactiva la atribución.
    binding.mapView.compass.fadeWhenFacingNorth = false // Brújula siempre activa.

    Log.d("Mapbox", "Map is ready")
}
```

A continuación, se añadirá la siguiente propiedad a la clase principal, la que se encarga de pintar el mapa.

```
private lateinit var permissionsManager: PermissionsManager
```

La clase `PermissionsManager` pertenece a la propia API de Mapbox, facilitando la gestión del permiso de localización, permisos que previamente se han añadido al *manifest*. En el método *onCreate()* de la clase principal se cargará la configuración del mapa y se comprobarán los permisos.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    onMapReady()

    // Gestión del permiso de localización.
    if (PermissionsManager.areLocationPermissionsGranted(this)) {
        Log.d("Mapbox", "Location permissions granted")
        activateLocation()
    } else {
        permissionsManager = PermissionsManager(permissionsListener)
        permissionsManager.requestLocationPermissions(this)
    }
}
```

El método `activateLocation()` se detallará más adelante, puedes dejarlo comentado, este se llamará si ya se dispone del permiso de localización, en caso contrario, deberá solicitarse implementándose la interface `PermissionsListener`.

```
private var permissionsListener: PermissionsListener = object : PermissionsListener {
    override fun onExplanationNeeded(permissionsToExplain: List<String>) {
        Toast.makeText(
            this@MainActivity,
            "Permissions: $permissionsToExplain",
            Toast.LENGTH_SHORT
        ).show()
    }

    override fun onPermissionResult(granted: Boolean) {
        if (granted) {
            Toast.makeText(
                    this@MainActivity, "Permissions granted", Toast.LENGTH_SHORT
```

```
            ).show()
            activateLocation()
        } else Toast.makeText(
            this@MainActivity,
            "Permissions not granted",
            Toast.LENGTH_SHORT
        ).show()
    }
}
```

En esta interface se sobrecargarán método `onExplanationNeeded()` para dar una explicación sobre la necesidad del permiso, el parámetro recibido es una lista con los permisos que necesitan explicación. Y el método `onPermissionResult()`, donde se podrá evaluar, a través del parámetro *granted* el estado del permiso.

También será necesario sobrecargar el método `onRequestPermissionsResult()` para recoger la respuesta del usuario al permiso y poder pasar dicha respuesta al *PermissionsManager*.

```
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<String>, // Al autocompletar, aparece como <out String>.
    grantResults: IntArray
) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    permissionsManager.onRequestPermissionsResult(requestCode, permissions, grantResults)
}
```

A continuación, se crea el método `activateLocation()` y dónde se implementarán dos interfaces para obtener el posición de la ubicación y mover la cámara al punto exacto. Recuerda activarlo si lo has comentado anteriormente.

```
// Listener para pasar la ubicación del usuario a la cámara.
private val onIndicatorPositionChangedListener = OnIndicatorPositionChangedListener {
    Log.d("Mapbox-Location", "Location: ${it.latitude()}, ${it.longitude()}")
    // Se actualiza la cámara con la ubicación del usuario.
    binding.mapView.mapboxMap.setCamera(CameraOptions.Builder().center(it).build())

    binding.mapView.gestures.focalPoint = binding.mapView
        .mapboxMap.pixelForCoordinate(it)
}

// Listener para activar la rotación de la cámara con el movimiento del usuario.
private val onIndicationPositionChangedListener = OnIndicatorBearingChangedListener {
    Log.d("Mapbox-Location", "Bearing: $it")
    // Se actualiza la cámara con la ubicación del usuario.
    binding.mapView.mapboxMap.setCamera(CameraOptions.Builder().bearing(it).build())
}

private fun activateLocation() {
    binding.mapView.location.enabled = true
```

```
    // Se pasa la ubicación del usuario a la cámara activando un tracker.
    binding.mapView.location.addOnIndicatorPositionChangedListener(
        onIndicatorPositionChangedListener
    )

    // Listener para activar la rotación de la cámara con el movimiento del usuario.
    binding.mapView.location.addOnIndicatorBearingChangedListener(
        onIndicationPositionChangedListener
    )
}
```

Por último, para evitar pérdidas de memoria al salir de la *activity* (o pasar a otra sin cerrar la aplicación), se recomienda desactivar los *listener* del seguimiento en el método `onStop()`.

```
override fun onStop() {
    super.onStop()
    binding.mapView.location.removeOnIndicatorPositionChangedListener(
        onIndicatorPositionChangedListener
    )

    binding.mapView.location.removeOnIndicatorBearingChangedListener(
        onIndicationPositionChangedListener
    )
}
```

Como resultado, se obtendría la ubicación del dispositivo, o el emulador, mediante el *puck* azul dibujado en el mapa.

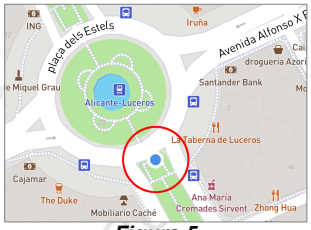

Figura 5

Además, el *puck* puede personalizarse, mediante un elemento personalizado utilizando *drawables*, o con pequeñas modificaciones, como por ejemplo, añadir un pulso sobre el *puck* para que destaque más sobre el mapa. Para esto último, puedes añadir las siguientes líneas en la activación de la localización.

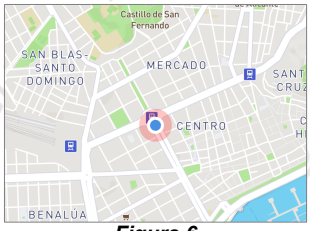

Figura 6

```
// Activación y personalización del puck.
binding.mapView.location.updateSettings {
    enabled = true
    pulsingEnabled = true
    pulsingColor = Color.RED
    pulsingMaxRadius = 60f
}
```

15.1.3. Captura de eventos en el mapa

El SDK de Mapbox ofrece una gran variedad de *listeners* para capturar cualquier evento[7] que se produzca sobre el mapa. A continuación, se tratarán los eventos más comunes, pero existen muchos más, así como otros muchos que dependerán del *plugin* de Mapbox que se esté utilizando en el proyecto.

El primer *listener* en verse será el **click sobre el mapa**, tanto un *click* simple (`addOnMapClickListener`) como una pulsación larga (`addOnMapLongClickListener`). En ambos casos, el *callback* devolverá un objeto `Point` con la información del punto pulsado. La ubicación más indicada para este tipo de *listeners* es en el método `onMapReady()`, de esta forma se asegura que el mapa esté cargado por completo.

```
val mapbox = binding.mapView.mapboxMap
...
mapbox.addOnMapClickListener { point ->
    Toast.makeText(
        this@MainActivity,
        "Lat: ${point.latitude()}\nLon: ${point.longitude()}",
        Toast.LENGTH_SHORT
    ).show()

    true
}
```

Observa como se obtienen los datos longitud y latitud del objeto *Point* recibido. Si quieres añadir el *listener* para una pulsación larga, bastará con sustituir el método utilizado en el ejemplo, `addOnMapClickListener` por `addOnMapLongClickListener`. El resultado será un *Toast* con cada pulsación indicando las coordenadas.

También se podría desplazar la cámara mediante, una animación, hacia el punto sobre el que se ha pulsado. Añade la siguiente línea al evento *onClick* tras el *Toast* para realizar el "vuelo".

```
mapbox.flyTo(moveCam(point), ANIM_CAM)
```

La función `moveCam()` y la variable `ANIM_CAM` contienen, el ajuste del zoom, la inclinación, rotación y punto central en el método y la duración de la animación.

```
private fun moveCam(punto: Point) = cameraOptions {
    zoom(12.0)
    pitch(60.0) // Inclinación.
    bearing(45.0) // Rotación.
    center(punto)
}

private val ANIM_CAM = MapAnimationOptions.mapAnimationOptions { duration(2000) }
```

7 Interacción con el usuario (https://docs.mapbox.com/android/maps/guides/user-interaction/)

Observarás que tras moverse la cámara a la nueva ubicación, volverá a la ubicación en tiempo real, esto es debido a que está activado el *listener* sobre la localización. Si no quieres que esto ocurra tienes dos opciones, no añadir el *listener* para la localización en tiempo real, o desactivar el *listener* antes de "volar", como se hace en el método *onStop()* visto anteriormente.

```
// Se elimina el tracking de la cámara sobre la posición del usuario para evitar
// que vuelva tras pulsar otro punto.
binding.mapView.location.removeOnIndicatorPositionChangedListener(
    onIndicatorPositionChangedListener
)
```

En ocasiones, puede que resulte más interesante obtener la posición del píxel que un objeto `Point`, para obtenerlo, por ejemplo en los eventos click donde se obtiene un objeto de clase *Point*, se podrían obtener las coordenadas del píxel con la siguiente línea.

```
val coordenadas = mapbox.pixelForCoordinate(point)
```

O si prefieres establecer un punto zoom, para que siempre se trabaje con la misma escala, puedes utilizar la siguiente línea.

```
val coordenadas = binding.mapView.mapboxMap.project(point, 20.0)
```

En ambos casos, una vez creada la variable `coordenadas`, se podría acceder a la X y la la Y usando `coordenadas.x` y `coordenadas.y`. El segundo parámetro que se pasa en `project()` es el factor de escala aplicado para obtener las coordenadas.

Existe otros eventos que puedes controlar cuando utilizas Mapbox, como por ejemplo, para saber cuando se mueva la cámara, en este caso, habrá que suscribirse mediante un *callback*.

```
mapbox.subscribeCameraChanged(cameraChangedCallback)
```

El *callback* `cameraChangedCallback` deberá definirse para realizar las acciones que se deseen, en este ejemplo simplemente se muestra una línea en el *log*.

```
private val cameraChangedCallback = CameraChangedCallback {
    Log.d("Mapbox", "Camera changed: $it")
}
```

En este caso el objeto ***it*** es de tipo ***CameraChanged***, del cual puede obtenerse información del estado de la cámara, como el zoom empleado, rotación, inclinación, etc.

Otro evento que puede resultar interesante es conocer cuando el usuario pulsa el icono de la brújula, por ejemplo, cada vez que se pulse la brújula se reinicia la cámara y se vuelve a activar el *tracking* sobre la ubicación real del dispositivo.

```
private val RESET_CAM = cameraOptions {
    zoom(9.0)
    pitch(0.0) // Inclinación.
    bearing(0.0) // Rotación.
}
...
```

```
binding.mapView.compass.addCompassClickListener {
    activateLocation()
    mapbox.flyTo(RESET_CAM, ANIM_CAM)
}
```

También es posible crear un *listener* para comprobar el movimiento del mapa, pero esta vez, el que realice el usuario, comprobando tres estados del movimiento, inicio, en movimiento y finalizado.

```
mapbox.addOnMoveListener(object : OnMoveListener {
    override fun onMove(detector: MoveGestureDetector): Boolean {
        Log.d("onMove", "Desplazándose.")
        return true
    }

    override fun onMoveBegin(detector: MoveGestureDetector) {
        Log.d("onMoveBegin", "Inicio del desplazamiento.")
    }

    override fun onMoveEnd(detector: MoveGestureDetector) {
        Log.d("onMoveEnd", "Fin del desplazamiento.")
    }
})
```

Esté se activará cuando el usuario desplace el mapa, el resultado para este código sería el siguiente en el *Logcat*.

```
2023..87  4780-4780  onMoveBegin    es.javiercarrasco.myfirstmapbox      D  Inicio del desplazamiento.
2023..06  4780-4780  onMove         es.javiercarrasco.myfirstmapbox      D  Desplazándose.
2023..20  4780-4780  onMove         es.javiercarrasco.myfirstmapbox      D  Desplazándose.
...
2023..85  4780-4780  onMove         es.javiercarrasco.myfirstmapbox      D  Desplazándose.
2023..97  4780-4780  onMove         es.javiercarrasco.myfirstmapbox      D  Desplazándose.
2023..98  4780-4780  onMoveEnd      es.javiercarrasco.myfirstmapbox      D  Fin del desplazamiento.
```

Existe toda una lista de eventos que pueden producirse durante el ciclo de vida del mapa, sobretodo durante su construcción. Los puedes consultar en la documentación de Mapbox. Algunos de los eventos producidos sobre los marcadores se verán a continuación.

15.1.4. Marcadores

Al igual que en otras APIs, Mapbox permite crear marcadores sobre un punto, pero también permite realizar círculos, polígonos o crear líneas entre diferentes puntos del mapa. En Mapbox estas acciones se conocen como anotaciones.

Mapbox no dispone de una marca por defecto para añadir anotaciones, por lo que deberás descargar[8] la marca que ellos te facilitan a través de la web.

Como comprobarás más adelante, crear una anotación (marcador) en Mapbox es algo más complejo que utilizando la API de *Google Maps*, pero no es algo que deba echarte para atrás.

8 Marca roja (https://docs.mapbox.com/android/maps/examples/red_marker.png)

Se comenzará creando la marca, como ya se ha dicho, deberás añadir la imagen del marcador como recurso *drawable*, a continuación, se creará un método que se encargue que añadir, de momento un marcador fijo. La llamada al método se hará desde el método `onMapReady()`.

```
private fun addAnnotation() {
    // Se crea una instancia de la API Annotation y se obtiene PointAnnotationManager.
    val annotationApi = binding.mapView.annotations
    val pointAnnotationManager = annotationApi.createPointAnnotationManager()
    val point = Point.fromLngLat(-0.5292, 38.4044)

    // Se configuran las opciones de la anotación.
    val pointAnnotationOptions = PointAnnotationOptions()
        // Se indica el punto de la anotación.
        .withPoint(point)
        // Se especifica la imagen asociada a la anotación.
        .withIconImage(
            BitmapFactory.decodeResource(
                this.resources,
                R.drawable.red_marker
            )
        )
        .withIconAnchor(IconAnchor.BOTTOM)

    // Se ajusta el tamaño de la marca.
    pointAnnotationOptions.iconSize = 0.5

    // Se añade el resultado al mapa.
    val pntAnnot = pointAnnotationManager.create(pointAnnotationOptions)
}
```

El resultado será el que se muestra en la figura, pero faltaría poder añadir un título para que se muestre la información del punto. Para añadir el título, como se ha comentado, el proceso es algo más complejo que con la API de Google Maps, pero no imposible. En primer lugar se creará un nuevo *layout* que contendrá el diseño del "globo" en el que mostrar el texto del título.

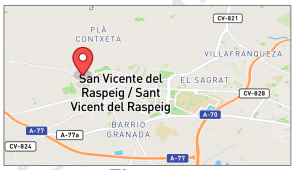

Figura 7

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:background="@drawable/border_title"
    android:orientation="vertical"
    android:padding="2dp">

    <TextView
        android:id="@+id/textViewTitulo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:gravity="center"
        android:padding="3dp"
        android:textAlignment="center"
        android:textAppearance="@style/TextAppearance.AppCompat.Body2"
        tools:text="Hello World" />
</LinearLayout>
```

Observa la propiedad *background* del *LinearLayout*, se utiliza un recurso *drawable* para establecer el color de fondo y un borde.

Ahora, deberás añadir al final del método `addAnnotation()` las siguientes líneas, estas se encargarán de representar la vista de la anotación, "inflando" el nuevo *layout* y ubicándola sobre la marca como una anotación, además de establecer en el *TextView* el texto a mostrar.

```
// Se prepara el título para la anotación.
val viewAnnotationManager = binding.mapView.viewAnnotationManager
val viewAnnotation = viewAnnotationManager.addViewAnnotation(
    resId = R.layout.annotation_layout,
    options = viewAnnotationOptions {
        annotationAnchor {
            geometry(point)
            anchor(ViewAnnotationAnchor.BOTTOM)
            // Corrección de la posición de la anotación.
            offsetY(((pntAnnot.iconImageBitmap?.height!! * pntAnnot.iconSize!!) + 10))
        }
    }
)
AnnotationLayoutBinding.bind(viewAnnotation)
    .textViewTitulo.setText(
        getString(
            R.string.txt_annotation,
            pnt.latitude().toString(),
            pnt.longitude().toString()
        ))
```

Puedes ver el título sobre la marca como se muestra en la figura. Puedes reajustar el método *addAnnotation()* de forma que el punto se pase por parámetro, junto con un texto para el título, para que cada vez que se llame cree una anotación, por ejemplo, al hacer una pulsación larga sobre el mapa.

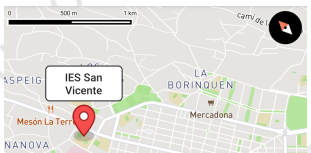

Figura 8

Pero esto plantea un nuevo problema, y es que puede llenarse la pantalla de marcadores. Una forma de solucionar este problema podría ser limpiando las marcas y los "globos" antes de crear una nueva anotación. Las dos siguientes líneas se encargarían de borrar la anotación y eliminar los "globos" creados.

```
binding.mapView.annotations.cleanup()
binding.mapView.viewAnnotationManager.removeAllViewAnnotations()
```

Puedes añadir un *listener* de tipo *onClick* sobre la *viewAnnotation*, por ejemplo, para hacer desaparecer el “globo”.

```
viewAnnotation.setOnClickListener {
    pointAnnotationManager.delete(pntAnnot)
    viewAnnotationManager.removeViewAnnotation(it)
}
```

15.1.5. Añadir UI al mapa

Si fuese añadir UI personalizada al mapa, se deberá empezar por editar la *activity* que lo contiene. En este caso, en el fichero `activity_main.xml` y se añadirá un FloatingActionButton en la parte inferior derecha de la pantalla.

La idea es que puedan añadirse tantas marcas como se quiera al mapa, sin eliminar las anteriores, el *floating button* añadido eliminará todas las marcas al pulsarse. En el método *onMapReady()* se añadirá el siguiente *listener* para el botón.

```
binding.floatingActionButton.setOnClickListener {
    binding.mapView.annotations.cleanup()
    binding.mapView.viewAnnotationManager.removeAllViewAnnotations()
}
```

El resultado de la modificación del fichero XML, y el *listener* sobre el nuevo botón añadido permite obtener el siguiente resultado.

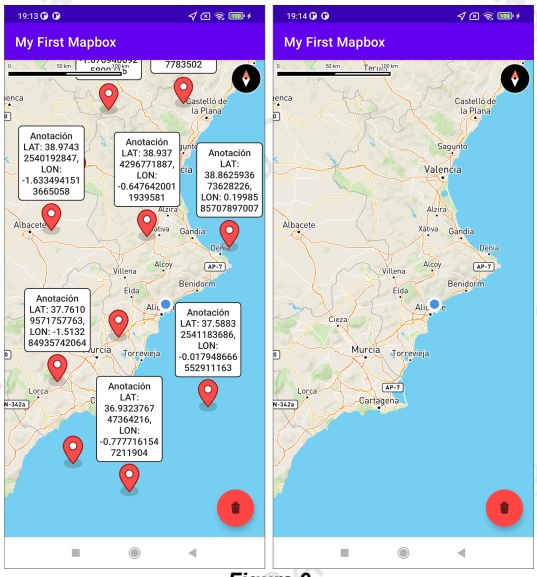

Figura 9

Puedes añadir cualquier tipo de vista sobre el mapa para interactuar con ella, como por ejemplo, un botón que te permita cambiar el estilo del mapa.

15.2. Google Maps

Ahora se tratará muy por encima la API de **Google Maps**. Esto se debe a que desde hace algunos años, Google obliga a introducir una tarjeta de crédito para poder hacer un uso completo de su API, y esto es algo que puede tirar para atrás a muchos de nosotros, sobretodo cuando se está aprendiendo. Pero, se intentará llegar hasta donde se pueda sin tener que dar esta información.

15.2.1. Obtener una API Key

El primer paso antes de comenzar a codificar será conseguir una API Key para poder hacer uso de la API de Google Maps. Para ello accede a la **Consola** de ***Google Cloud***[9] con una cuenta de Google. Una vez iniciada sesión, deberás crear un **Proyecto nuevo**.

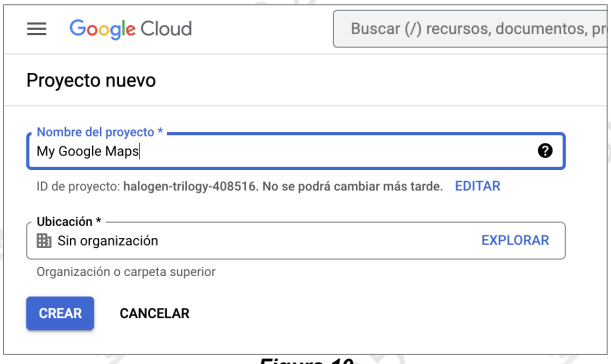

Figura 10

Una vez creado el proyecto (puedes utilizar el creado para Firebase) selecciona la opción **APIs y servicios**.

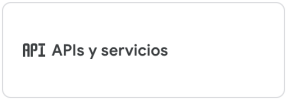

Figura 11

Una vez dentro, en el panel izquierdo tendrás varias opciones, la primera, **APIs y servicios habilitados**, mostrará las APIs activadas, si no tienes la que interesa para este punto (Maps SDK for Android[10]), deberás activarla utilizando la opción **Habilitar APIs y servicios** o desde la **Biblioteca**.

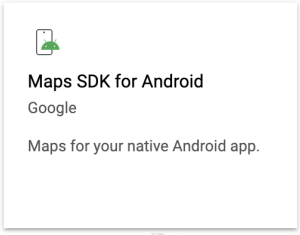

Figura 12

9 Google Cloud (https://console.cloud.google.com)
10 SDK de Maps para Android (https://developers.google.com/maps/documentation/android-sdk)

Selecciona la opción y una vez dentro, habilita la API. Para finalizar la activación piden datos bancarios, una tarjeta de débito o crédito, este paso deberás activarlo bajo tu responsabildiad.

Una vez habilitado, deberás crear una **API Key** desde la opción **Claves y credenciales** del panel izquierdo. Además, también se recomienda la aplicación de restricciones para proteger el uso fraudulento.

Como puedes ver en la configuración, se indica el tipo de restricción, en este caso, para **Apps de Android**, además deberá indicarse el **nombre del paquete** y el **certificado SHA-1** del desarrollador. Si desarrollas en más de un equipo la misma aplicación, deberás añadir tantas aplicaciones como sea necesario con el mismo nombre.

Para obtener el certificado SHA-1 del equipo, la forma más rápida y sencilla es mediante **Gradle**. Desde el panel lateral derecho, abre la opción Gradle y ejecuta la opción ***Nombre_proyecto > Task > signingReport***.

Figura 13

Automáticamente aparecerá en la parte inferior el resultado con el certificado. Recuerda volver a la opción **app** la ejecución del IDE.

15.2.2. Configuración

Para comenzar a utilizar Google Maps, se iniciará un proyecto nuevo mediante una *Empty Views Activity*. Seguidamente, se terminará de configurar el proyecto, debiendo indicarse el nombre del proyecto, *My First Google Maps*, y la API mínima (26). Una vez cargado, se terminará de configurar Google Maps en el proyecto[11].

Será necesario añadir la dependencia de Servicios de Google Play para el SDK de Maps para Android.

```
implementation("com.google.android.gms:play-services-maps:18.2.0")
```

Ahora se añadirá la clave para la API, para hacerlo de forma segura se recomienda instalar el complemento *Secrets Gradle*, principalmente, la idea es dejar la *key* fuera del control de versiones, como GitHub. En este caso no se añadirá este complemento, se hará de forma similar a la vista con Mapbox. Se creará el recurso `google_maps_api.xml` en *values*, y se añadirá la clave para la API.

```
<resources>
    <string name="google_maps_key" templateMergeStrategy="preserve"
        translatable="false">YOUR_API_KEY</string>
</resources>
```

Recuerda sacar del control de versiones este fichero, si utilizas GitHub, deberás incluirlo en el fichero **.gitignore**.

11 Configura un proyecto de Android Studio (https://developers.google.com/maps/documentation/android-sdk/config)

Como puedes observar en el recurso que se acaba de añadir, se han utilizado dos parámetros que no se habían visto hasta ahora.

- ***templateMergeStrategy***, esta propiedad indica al motor de plantillas de Android Studio que esta línea no debe modificarse cuando se cree una nueva actividad de Google Maps, conservando así la clave.
- ***translatable***, este simplemente indica que es un recurso que no debe traducirse.

Una vez añadida la clave, debes indicar la clave en el fichero *Manifest*. Para ello añade la siguiente *meta-data* dentro de la etiqueta *application*.

```
<application
    ...>
    <meta-data
        android:name="com.google.android.geo.API_KEY"
        android:value="@string/google_maps_key" />
```

Ahora, en ***activity_main.xml*** se añadirá de forma estática la vista que representará el mapa, se hará haciendo uso del sistema "moderno", utilizando ***SupportMapFragment***, que permite administrar el ciclo de vida de un objeto *GoogleMap*.

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

En este punto, si lanzas la aplicación, ya verás el mapa representado en la actividad. Para poder interactuar con el mapa, habrá que conseguir el *handle* del *fragment*, indicándolo el ID del recurso que contiene el mapa.

También habrá que extender la actividad con la interfaz ***OnMapReadyCallback***, con esta se podrá registrar la devolución de la llamada en el *fragment*.

```
class MainActivity : AppCompatActivity(), OnMapReadyCallback {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        val mapFragment =
            supportFragmentManager.findFragmentById(binding.map.id) as SupportMapFragment?
        mapFragment?.getMapAsync(this)
    }

    override fun onMapReady(googleMap: GoogleMap) {
```

```
        googleMap.addMarker(
            MarkerOptions()
                .position(LatLng(0.0, 0.0))
                .title("Marker")
        )
    }
}
```

Al implementar la interfaz *OnMapReadyCallback* debe sobrecargarse el método ***onMapReady***, donde se recoge el objeto *GoogleMap* y puede interactuarse con él, observa como se añade una marca en el mapa, algo más sencillo que con Mapbox.

15.2.3. Localización y permisos

Con lo visto hasta ahora, podrás comprobar que al lanzar la aplicación marcará en el mapa la ubicación (0.0, 0.0). Ahora, se puede crear un objeto *LatLng* para establecer la marca en Alicante (38.345996, -0.490686) por ejemplo. Modifica el título del marcador para que aparezca el nombre de la ubicación al pulsar sobre él.

```
val alicante: LatLng = LatLng(38.345996, -0.490686)

googleMap.addMarker(
    MarkerOptions()
        .position(alicante)
        .title("Alicante")
)

googleMap.moveCamera(CameraUpdateFactory.newLatLngZoom(alicante, 10f))
```

La primera línea crea un objeto de tipo *LatLng* con las coordenadas del sitio. El método ***addMarker*** crea la marca sobre el mapa con la posición y el título. Por último, el método ***moveCamera*** se encarga de mover la cámara hacia la posición de la marca. Tras ampliar el mapa y pulsar sobre el marcador, podrás observar la nueva posición y el título indicado. Puedes emplear el método ***animateCamera*** en lugar de *moveCamera*, este añadirá una animación en el movimiento.

Figura 14

El siguiente paso será añadir los permisos para utilizar la localización al fichero `AndroidManifest.xml`.

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

Además, para poder hacer uso de los servicios de localización será necesario añadir la siguiente dependencia al fichero *Gradle(:app)*.

```
implementation("com.google.android.gms:play-services-location:21.0.1")
```

De vuelta al ejemplo, lo que se ha hecho en realidad es añadir un marcador al mapa en un punto concreto, en el siguiente apartado se ampliará este concepto. A continuación, se verá como realizar el posicionamiento, como ya está configurada la ubicación en el emulador (no necesario en un dispositivo físico), se podrán añadir varias y ver como estas van cambiando según interese.

Será necesario solicitar permiso al usuario para que permita a la aplicación acceder a la ubicación del dispositivo, en este caso se dejará que el sistema gestione el código de respuesta. Se comenzará modificando las propiedades de la clase, ya pensando en próximas actualizaciones.

```
class MainActivity : AppCompatActivity(), OnMapReadyCallback {
    private lateinit var fusedLocationProviderClient: FusedLocationProviderClient
    private lateinit var lastLocation: Location
    private lateinit var map: GoogleMap
    ...
```

Estas declaraciones son, el objeto ***fusedLocationProviderClient*** de clase `FusedLocationProviderClient` que permitirá recoger información referente a la ubicación y el objeto ***lastLocation*** de clase `Location` para almacenar la posición anterior. El objeto ***map*** se creará para la gestión del mapa, se instanciará en el método *onMapReady*, así podrá utilizarse en cualquier momento. También se creará el siguiente objeto para capturar la respuesta a la solicitud del permiso.

```
private val requestPermissionLauncher =
    registerForActivityResult(ActivityResultContracts.RequestPermission()) { isGranted ->
        if (isGranted) {
            // Permission is granted. Continue the action or workflow in your app.
            Log.i("PERMISOS", "Permiso concedido.")
            configMap()
        } else {
            Log.e("PERMISOS", "Explicación de la necesidad del permiso.")
            finish()
        }
    }
```

En primer lugar, en el método ***onCreate***, tras obtener el *supportFragmentManager*, se añadirá la siguiente línea para inicializar el objeto *fusedLocationProviderClient*.

```
fusedLocationProviderClient = LocationServices.getFusedLocationProviderClient(this)
```

A continuación, se creará el método ***configMap***, este método servirá para poder controlar la gestión del permiso y la obtención del posicionamiento según la ubicación del dispositivo (real o emulado).

```
// Comprueba el permiso de ubicación y recoloca el mapa según la ubicación.
@SuppressLint("MissingPermission")
private fun configMap() {
    when {
        (isPermissionGranted()) -> {
            // Se añade la marca en la ubicación real.
            map.isMyLocationEnabled = true
```

```
                fusedLocationProviderClient.lastLocation.addOnSuccessListener { location ->
                    if (location != null) {
                        lastLocation = location
                        val currentLatLng = LatLng(location.latitude, location.longitude)
                        map.animateCamera(CameraUpdateFactory.newLatLngZoom(currentLatLng, 10f))
                    }
                }
            }

            else -> {
              requestPermissionLauncher.launch(android.Manifest.permission.ACCESS_FINE_LOCATION)
            }
        }
    }
```

En este fragmento de código, como puedes observar, se comprueba en primer lugar el estado del permiso haciendo uso del método ***isPermissionGranted()***, que se detalla a continuación. Si el permiso está concedido, se activa la ubicación con ***isMyLocationEnabled()***. Seguidamente se crea un *listener* sobre la última localización del objeto *fusedLocationProviderClient* para comprobar la ubicación, se almacenará y moverá la cámara al punto concreto. Si no está concedido el permiso, se solicitará haciendo uso de *requestPermissionLauncher*.

```
// Comprueba el estado del permiso.
private fun isPermissionGranted() = ContextCompat.checkSelfPermission(
    this, android.Manifest.permission.ACCESS_FINE_LOCATION
) == PackageManager.PERMISSION_GRANTED
```

Por último, el método ***onMapReady()*** podría quedar de la siguiente forma.

```
override fun onMapReady(googleMap: GoogleMap) {
    map = googleMap

    // Se habilitan los botones del zoom.
    map.uiSettings.isZoomControlsEnabled = true
    // Se habilita la brújula, solo aparecerá cuando se gire el mapa.
    map.uiSettings.isCompassEnabled = true
    // Se habilita el botón para centrar la ubicación actual (por defecto es true).
    map.uiSettings.isMyLocationButtonEnabled = true

    configMap()
}
```

Si lanzas la aplicación, verás como pregunta por la concesión del permiso para utilizar la ubicación del dispositivo. Al conceder el permiso, podrás ver que re-coloca el mapa automáticamente y ubicará el dispositivo directamente, mediante el típico punto azul, en la posición en la que se encuentre.

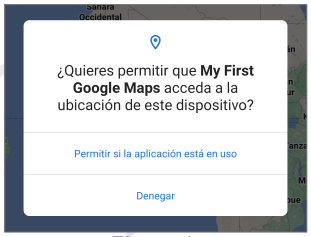

Figura 15

La ventana que aparece en la figura variará según la API del sistema operativo y otras configuraciones que puedas tener.

Con esto, ya tendrías el posicionamiento activado haciendo uso de la API de *Google Maps*. Observa como aparece el botón “Ubicación” en la esquina superior derecha de la pantalla, se activa por defecto al usar el posicionamiento. Fíjate también que se han activado los botones de zoom y la brújula, que aparecerá solo cuando se gire el mapa.

Figura 16

Es posible que en el emulador, la primera vez no sea capaz de pintar el punto azul para indicar la ubicación (esto no ocurre en los dispositivos físicos), tras un segundo inicio ya suele aparecer.

15.2.4. Captura de eventos en el mapa

Si se necesita saber cuando se pulsa el botón “Ubicación” o “Centrar” para realizar alguna acción concreta, deberá extenderse la clase la clase principal con `GoogleMap.OnMyLocationButtonClickListener`. Además, se creará en el método `onMapReady()` el *listener* sobre dicho botón añadiendo la siguiente línea antes de llamar al método `configMap()`.

```
map.setOnMyLocationButtonClickListener(this)
```

También deberá sobrecargarse el método ***onMyLocationButtonClick***, en el que se realizarán las acciones que interesen. Deberás tener en cuenta que al sobrecargar este método deberás controlar la vuelta a la ubicación por ti mismo si devuelves *true*, pero si devuelves *false*, volverá a la ubicación tras realizar las acciones.

```
override fun onMyLocationButtonClick(): Boolean {
    Log.d("onMyLocationButtonClick", "Click sobre el botón ubicación!!")
    return false
}
```

Si lo que se quiere es controlar la pulsación sobre la misma ubicación, es decir, sobre el punto azul, deberá extenderse la clase `GoogleMap.OnMyLocationClickListener` en la clase principal, y activar el *listener* sobre el punto azul con la siguiente línea:

```
map.setOnMyLocationClickListener(this)
```

También deberá sobrecargarse el siguiente método en el que se indicarán las acciones a realizar al producirse la pulsación sobre la ubicación.

```
override fun onMyLocationClick(p0: Location) {
    Log.d("onMyLocationClick", "Mi ubicación pulsada [${p0.latitude}, ${p0.longitude}]")
}
```

Si lo que se desea es saber en que punto del mapa se ha pulsado, se utilizará el método `setOnMapClickListener()`, este método hace uso del parámetro `LatLng`, representado por la variable `it` en el siguiente fragmento de código, que indicará la ubicación pulsada en el mapa.

```
map.setOnMapClickListener { it: LatLng
    Log.d("onMapClickListener", it.toString())
    map.animateCamera(CameraUpdateFactory.newLatLng(it))
```

```
}
```

Con este fragmento de código, cada vez que se pulse sobre el mapa, se verá en el *Logcat* una línea similar a la siguiente, además del desplazamiento de cámara.

```
2023-... onMapClickListener  es.javiercarrasco.myfirstgooglemaps D lat/lng: (38.3686351,-0.42269904)
```

Puedes ahorrarte la extensión de la clase principal y aplicar *lambdas* para capturar los eventos como se ha hecho para *setOnMapClickListener*, por ejemplo.

```
map.setOnMyLocationClickListener { it: Location
    Log.d("onMyLocationClick", "Mi ubicación pulsada [${it.latitude}, ${it.longitude}]")
}
```

Existen más eventos que pueden producirse sobre un mapa, para conocer más, visita la sección *Como interactuar con un mapa* de la documentación de la API[12].

15.2.5. Marcadores

Siguiendo con el ejemplo, a continuación se añadirá una serie de marcadores al mapa de forma dinámica (por código). Este es un proceso muy sencillo, en primer lugar, se creará un método para configurar el marcador en sí.

```
// Método para añadir la marca en la localización indicada.
private fun placeMarker(location: LatLng) {
    Log.d("placeMarker", "Marcador en la ubicación: $location")

    // Se crea un objeto MarkerOptions para configurar la marca.
    val markerOptions = MarkerOptions().position(location)

    markerOptions.title("Mi marca")  // Título de la marca.
    markerOptions.snippet("Esta es mi marca")  // Descripción de la marca.

    // Se añade la marca al mapa.
    map.addMarker(markerOptions)
}
```

Se crea el objeto `MarkerOptions` para establecer la configuración del marcador y añadirlo al mapa. En este caso, la modificación que se ha hecho sobre el marcador es asignarle la ubicación en la que debe aparecer y se añade una descripción (*snippet*). A continuación, para conseguir que cada vez que se pulse sobre una parte del mapa, se añada un marcador, deberá llamarse al método en cuestión. Será necesario pasar la posición, para eso se utilizará el parámetro `LatLng`.

```
map.setOnMapClickListener { it: LatLng
    Log.d("onMapClickListener", it.toString())
    placeMarker(it)
    map.animateCamera(CameraUpdateFactory.newLatLng(it))
}
```

12 Maps SDK for Android (https://developers.google.com/maps/documentation/android-sdk/intro)

Si pruebas este código, verás que a cada *click* en el mapa aparece un pin, y esta quizá no sea la mejor opción para navegar por un mapa. En tal caso, se puede optar por hacer aparecer un marcador tras una pulsación larga sobre un punto concreto del mapa. Bastaría con sustituir `setOnMapClickListener` por `setOnMapLongClickListener`.

Figura 17

El resultado lo puedes ver en la figura, dónde puedes ver que aparece el típico pin rojo de Google Maps al realizar una pulsación larga sobre cualquier punto del mapa.

Si quieres cambiar el color del pin para que no sea el típico pin rojo, puedes utilizar la siguiente instrucción.

```
// Color de la marca.
markerOptions.icon(BitmapDescriptorFactory.defaultMarker(
        BitmapDescriptorFactory.HUE_VIOLET
    )
)
```

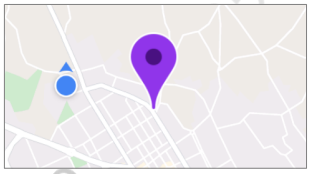

Figura 18

A continuación, se verá como saber que se ha pulsado un marcador, cual, y como obtener algo más de información sobre la marca en concreto.

Para saber que se ha pulsado sobre un marcador, se puede extender la clase principal con `GoogleMap.OnMarkerClickListener`, añadiéndola a la declaración de la clase principal, esto obligará a implementar el método `onMarkerClick()`, o cómo se ha visto anteriormente, implementar la siguiente *lambda*.

```
map.setOnMarkerClickListener { marker ->
    Log.d("onMarkerClickListener", marker.id)
    false
}
```

Ahora ya está activada la escucha sobre los marcadores. A continuación, se obtendrá la información, concretamente, la dirección sobre la que se ha añadido el marcador.

Para obtener una dirección, se debe hacer uso de la clase `Geocoder`, que devolverá una lista de direcciones de la posición indicada. Se añadirá el siguiente método a la clase para realizar esta operación.

```
// Método para obtener la dirección de una ubicación.
private fun getAddress(location: LatLng): String {
    // Se crea un objeto Geocoder para obtener la dirección.
    val geocoder = Geocoder(this)
    // Se pueden obtener más de una dirección de un mismo punto.
    val addresses: MutableList<Address>?
    val address: Address?
    var addressText = ""

    // Se obtiene la información del punto concreto.
```

```
    if (Build.VERSION.SDK_INT < 33) {
        addresses = geocoder.getFromLocation(location.latitude, location.longitude, 1)

        // Si se obtiene la dirección, se añade a la variable addressText.
        if (!addresses.isNullOrEmpty()) {
            // Se coge la primera posición.
            address = addresses[0]
            // Se comprueba que la dirección tenga o no más de una línea.
            if (address.maxAddressLineIndex > 0) {
                for (i in 0 until address.maxAddressLineIndex) {
                    addressText += if (i == 0) address.getAddressLine(i)
                    else "\n${address.getAddressLine(i)}"
                }
            } else { // Acción más habitual.
                if (!address.thoroughfare.isNullOrBlank())
                    addressText += address.thoroughfare
                if (!address.subThoroughfare.isNullOrBlank())
                    addressText += ", ${address.subThoroughfare}"
            }
        } else addressText = "no address found"
    } else addressText = "no address found"

    return addressText
}
```

En definitiva, este método se encargará de formatear y devolver en un *string* una dirección. En el método ***placeMarker***, dónde se crean los marcadores, se hará uso de la opción *title* para añadir la información que se mostrará al pulsar sobre una marca, esto justo antes de añadir el marcador al mapa.

```
// Título de la marca.
markerOptions.title(getAddress(location))
```

Para hacer aparecer sobre la marca el globo que contendrá la información de la dirección del punto al pulsarla, deberá añadirse la siguiente línea al método `onMarkerClick()`.

Figura 19

```
marker.showInfoWindow()
```

El globo deberá aparecer al pulsar sobre las marcas y desaparecerá al pulsar sobre cualquier otro punto del mapa.

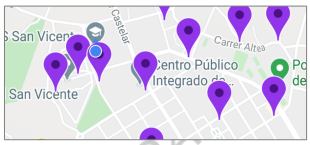

Figura 20

El problema de los marcadores es que si no se lleva un control, es posible encontrarse con un mapa completamente lleno de marcadores creados por el usuario.

Una forma fácil de solucionar este problema podría ser que, cada vez que se añada un marcador, se elimine el anterior. Para eso, se creará la siguiente propiedad en la clase principal.

```
private var lastMarker: Marker? = null
```

Ahora, lo que deberá hacerse es modificar el método ***placeMarker*** para que quede de la siguiente forma.

```
// Se elimina la última marca.
if (lastMarker != null)
    lastMarker!!.remove()

// Se añade la marca al mapa y se guarda en lastMarker.
lastMarker = map.addMarker(markerOptions)
```

De esta forma, cada vez que se añada un nuevo marcador se eliminará el anterior. Si se necesita poder añadir varios, se deberá controlar los marcadores añadidos mediante una lista, de esta manera estarán todos controlados.

También se podría eliminar los marcadores pulsando sobre ellos mismos, pero deberás tener en cuenta que no llegarías a mostrar la información si la tuviese, ya que los destruirías antes. Este método sería extendiendo de `GoogleMap.OnMarkerClickListener`.

```
override fun onMarkerClick(p0: Marker?): Boolean {
    Log.d("onMarkerClick", "Click sobre una marca")
    p0!!.remove()
    return false
}
```

15.2.6. Añadir UI al mapa

Como se vio inicialmente en esta sección, se ha añadido un *FragmentContainerView* para poder mostrar un mapa en la actividad.

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

Para añadir elementos UI al mapa, deberá añadirse un elemento de tipo *ViewGroup* que contendrá este *FragmentContainerView* y todos los elementos que se quieran utilizar, como por ejemplo, un botón.

Se dejará el *layout* como se ve a continuación, se añadirá un *RelativeLayout* que contendrá el *FragmentContainerView* para el mapa (*+id/map*), y un botón que aparecerá por encima en la esquina superior izquierda.

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

```
    android:layout_height="match_parent">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/map"
        android:name="com.google.android.gms.maps.SupportMapFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <com.google.android.material.button.MaterialButton
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentStart="true"
        android:layout_alignParentTop="true"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:text="@string/txt_type_normal" />
</RelativeLayout>
```

A continuación, se añadirán las siguientes líneas de código al método ***configMap***, justo tras la activación de la ubicación. Además, se verá como cambiar el tipo, o estilo, de mapa que se quiera mostrar.

```
// Se establece el tipo de mapa.
map.mapType = GoogleMap.MAP_TYPE_NORMAL // Por defecto.
binding.button.text = getString(R.string.txt_type_normal)
```

En el método ***onMapReady*** se añadirá el *listener* para el botón, de forma que permita ir cambiando el tipo de mapa en cada pulsación.

```
binding.button.setOnClickListener { // Se selecciona el tipo de mapa.
    when (map.mapType) {
        GoogleMap.MAP_TYPE_NORMAL -> {
            map.mapType = GoogleMap.MAP_TYPE_HYBRID
            binding.button.text = getString(R.string.txt_type_hybrid)
        }
        GoogleMap.MAP_TYPE_HYBRID -> {
            map.mapType = GoogleMap.MAP_TYPE_SATELLITE
            binding.button.text = getString(R.string.txt_type_satellite)
        }
        GoogleMap.MAP_TYPE_SATELLITE -> {
            map.mapType = GoogleMap.MAP_TYPE_TERRAIN
            binding.button.text = getString(R.string.txt_type_terrain)
        }
        GoogleMap.MAP_TYPE_TERRAIN -> {
            map.mapType = GoogleMap.MAP_TYPE_NORMAL
            binding.button.text = getString(R.string.txt_type_normal)
        }
    }
}
```

El resultado de todo este código puedes verlo representado en la siguiente figura, fíjate que se actualiza el texto del botón a la vez que cambia el tipo del mapa, indicando el nombre del tipo de mapa mostrado.

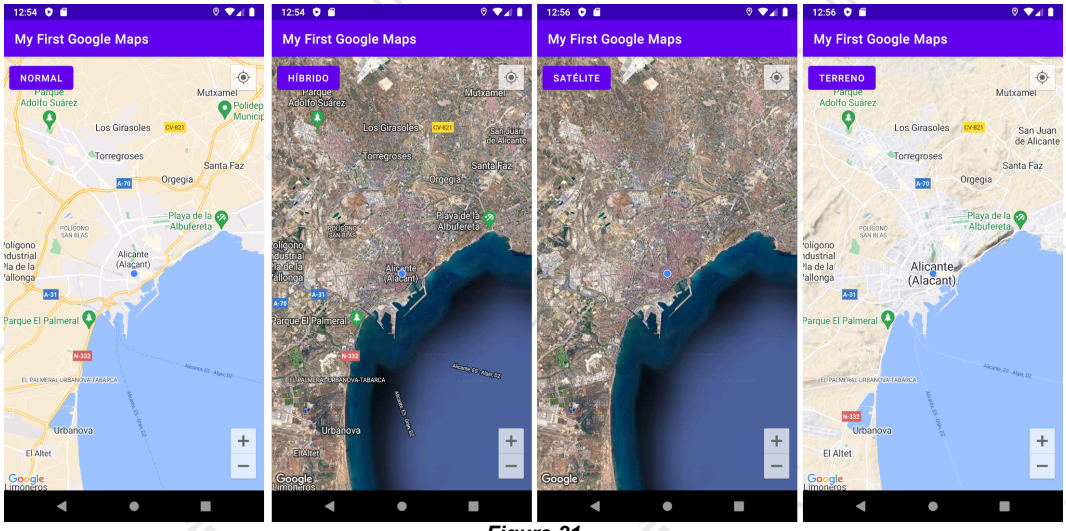

Figura 21

Con esto ya dispones de suficiente conocimiento para representar información mediante la API de *Google Maps*, se puede hacer mucho más, pero parte de ello requiere el pago de una cuota.

Además de las dos APIs vistas para mapas (Mapbox[13] y Google Maps[14]) existen otras como HERE Maps API[15] o MapTiler[16].

13 Mapbox (https://www.mapbox.com/)
14 Google Maps (https://developers.google.com/maps)
15 HERE Maps API (https://www.here.com/)
16 MapTiler (https://maptiler.es/)