

Programación Multimedia y Dispositivos Móviles (UP6) (Parte 2)

Autores: Antonio Calabuig Puigvert y Sebastián Villa Ponce

Centro: IES Salvador Gadea

Departamento: Informática

Ciclo: Desarrollo de Aplicaciones Multiplataforma

Fecha: 10/01/2026

1. Introducción y base teórica
 2. Estilos previos
 3. Expo - MapView
 4. Configuración Google Cloud
 5. Marcadores
 6. Acciones para seguir y ver la ubicación del usuario
 7. Creación de un CustomMap
 8. Ubicación del usuario y seguir movimiento
 9. Mover la cámara para seguir al usuario
 10. FAB para personalizar acciones
 11. Acciones con el mapa
 12. Polylines
 13. Uso de AsyncStorage para datos no confidenciales
-

1. Introducción y base teórica

- Marcadores
- Polylines
- Seguimiento de coordenadas
- Obtener la ubicación
- Controlar el mapa

2. Estilos previos

Antes de iniciar con los servicios de Maps vamos a preparar unos estilos en el fichero `index.tsx` de la carpeta `map`.

```
import { StyleSheet } from 'react-native'  
import { SafeAreaView } from 'react-native-safe-area-context';  
  
const MapScreen = () => {  
  return (  
    <SafeAreaView style={styles.container}>  
      <SafeAreaView style={styles.map}/>  
    </SafeAreaView>  
  )  
}
```

```

}

export default MapScreen;

const styles = StyleSheet.create({
  container: {
    flex: 1
  },
  map: {
    width: '100%',
    height: '100%',
    //backgroundColor: 'red'
  }
})

```

3. Expo - MapView

Ahora que tenemos una estructura visual básica, empezamos con la visualización del mapa.

Para poder trabajar con el API de Google necesitamos crear una `prebuild`, hasta llegar a ese punto trabajaremos con las visualizaciones que nos permite Expo con `MapView`. A continuación se encuentra el enlace a la documentación oficial:

The screenshot shows the official documentation for the `react-native-maps` library on the `expo.dev` website. The page title is "react-native-maps". It includes a brief description: "A library that provides a Map component that uses Google Maps on Android and Apple Maps or Google Maps on iOS." Below the description is a link: "https://docs.expo.dev/versions/latest/sdk/map-view/". On the right side of the page, there is a small logo for "react-native-maps" and the `expo` logo.

1. Empezamos con el comando de instalación:

```
npx expo install react-native-maps
```

2. Cambiamos nuestra vista interna del componente al tipo `MapView` :

```

import { StyleSheet } from 'react-native'
import { SafeAreaView } from 'react-native-safe-area-context';
import MapView from 'react-native-maps'

const MapScreen = () => {
  return (
    <SafeAreaView style={styles.container}>
      <MapView style={styles.map}/>
    </SafeAreaView>
  )
}

export default MapScreen;

const styles = StyleSheet.create({
  container: {
    flex: 1
  },
  map: {
    width: '100%',
  }
})

```

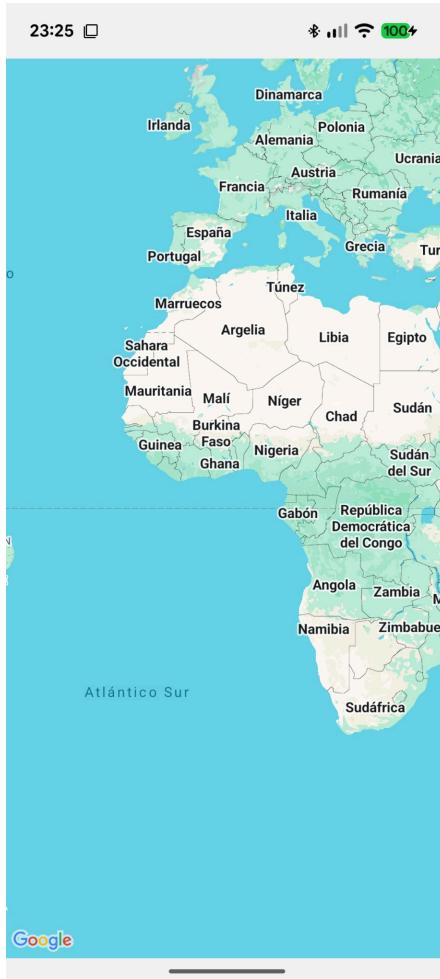
```

height: '100%',
//backgroundColor: 'red'
}
})

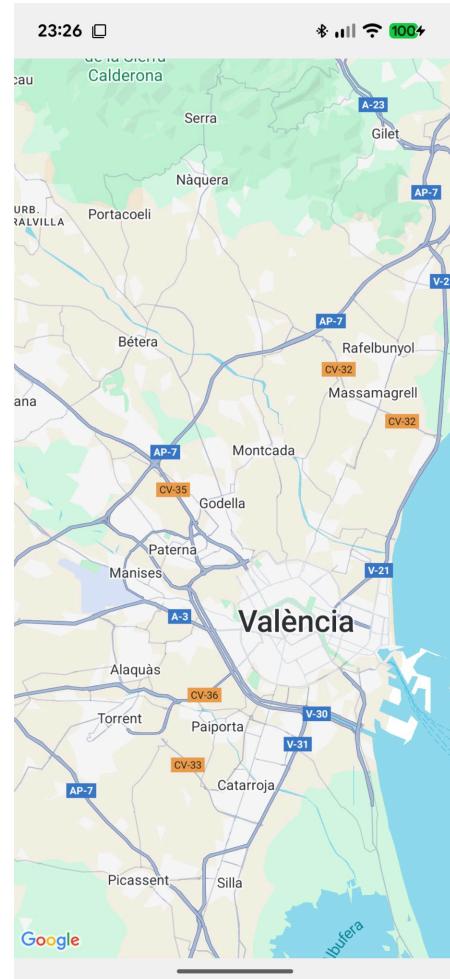
```

3. Ya deberíamos ser capaces de mostrar un mapa interactivo:

Mapa inicial



Mapa ampliado



Este mapa no está disponible en modo producción, solo en desarrollo para poder previsualizar un resultado de lo que sería trabajar con el API de Google

4. El componente `<MapView>` tiene atributos de configuración muy variados, algunos que usamos para mostrar sus uso son:

- `showsMyLocationButton` : que muestra el botón para ir a la ubicación del dispositivo
- `showsUserLocation` : que muestra el punto azul donde se sitúa el dispositivo móvil
- `initialRegion` : recibe un objeto con `latitude, longitude, latitudeDelta y longitudeDelta` para configurar un sitio por defecto inicial, todos los parámetros son obligatorios

```

const MapScreen = () => {
  return (
    <SafeAreaView style={styles.container}>

```

```

<MapView style={styles.map}
showsMyLocationButton
showsUserLocation

initialRegion={{
latitude: 39.459520762987665,
longitude: -0.47044131140655987,
latitudeDelta: 0,
longitudeDelta: 0
}}
/>
</SafeAreaView>
)
}
export default MapScreen;

```

4. Configuración Google Cloud

Ya estamos listos para sustituir la versión provisional del mapa por los servicios de Google Maps que vamos a consumir. Para ello seguimos los siguientes pasos:

1. Vamos al enlace y nos autenticamos con una cuenta de Google para poder crear un nuevo proyecto con sus credenciales:

<https://console.developers.google.com/apis>

2. Creamos un nuevo proyecto y nos saldrá una ventana similar a esta:

The screenshot shows the 'Create Project' dialog box from the Google Cloud Platform. The project name is set to 'My Project 71178'. The organization is listed as 'iesgadea.es'. The location is also 'iesgadea.es'. At the bottom, there is a 'Create' button.

Mi organización es iesgadea.es porque es un correo corporativo

3. Ponemos de proyecto **Expo Project** en caso de no tener ninguno creado, sino podemos usar el anterior para aprovechar el número limitado de proyectos gratuitos.

☰ Google Cloud

Proyecto nuevo

Nombre del proyecto * — [?](#)

ID del proyecto: expo-project-484816. No se puede cambiar más adelante. [Editar](#)

Organización * — [?](#)

Selección una organización para vincularla a un proyecto. No podrás cambiar esta selección más adelante.

Ubicación * — [Explorar](#)

Organización o carpeta superior

[Crear](#) [Cancelar](#)

4. Damos al apartado **Seleccionar proyecto** una vez creado en la parte superior derecha.
5. Seguimos los pasos de la documentación oficial donde nos dice que el primer paso es habilitar **Maps SDK para Android**. Buscamos en la barra de recursos y ponemos Maps SDK for Android y lo seleccionamos.

The screenshot shows the Google Cloud Platform API library interface. On the left, there's a sidebar with project navigation and a main area for 'APIs y servicios'. On the right, a search bar at the top says 'Maps SDK'. Below it, under 'Resultados principales', there are three items: 'Maps SDK for Android' (with a warning icon), 'Maps SDK for iOS', and 'Maps 3D SDK for Android'. Each item has a brief description and a link. Below these, under 'Instructivos y documentación', there are links to 'Google Maps' and 'Google Maps APIs Premium Plan: Included APIs'. Under 'Marketplace', there are links to 'Maps SDK for iOS', 'Maps SDK for Android', and 'Maps 3D SDK for iOS'. A note at the bottom says 'Se muestran resultados de recursos para Expo Project únicamente'.

6. Habilitamos el servicio.

 **Maps SDK for Android**
[Google](#)

Maps for your native Android app.

[Habilitar](#)

7. En caso de que nos salga tenemos que completar el proceso de autenticación. (OPCIONAL)



Paso 1 de 2 Información de la cuenta

 SEBASTIAN VILLA PONCE
svilla@iesgadea.es [Cambiar de cuenta](#)

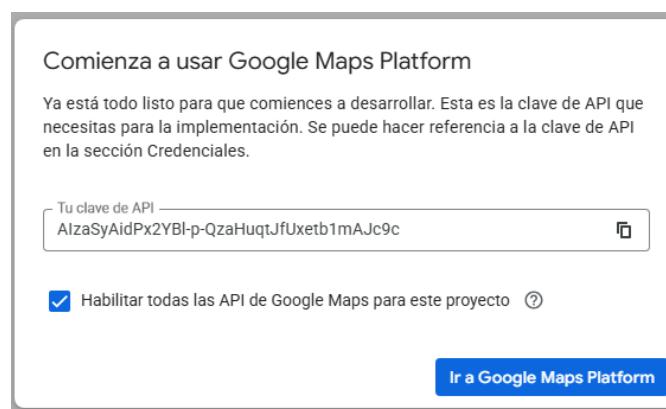
País

España ▾

Cuando usas esta aplicación, aceptas las Condiciones del Servicio de [Google Cloud Platform](#), la [Prueba gratuita complementaria](#) y [cualquier servicio y APIs aplicables](#).

[Aceptar y continuar](#)

8. Despues nos saldrá una ventana con nuestra clave API para el proyecto.



```
// Clave API de Sebas en caso de emergencia  
AlzaSyAidPx2YBl-p-QzaHuqtJfUxetb1mAJs9c
```

9. Copiamos la clave y seleccionamos Ir a [Google Maps Platform](#) y le damos a [Tal vez más tarde](#) para no restringir de momento nuestra clave.

10. Estamos en el Panel general donde debemos ir a [Claves y credenciales](#) :

11. Desde esta ventana se pueden realizar un conjunto de acciones y se puede acceder a las credenciales:

12. Vamos a comprobar como se edita la configuración de la clave y se limita su uso editando la clave en el menú de la derecha.

13. En caso de no tener un paquete creado en el fichero `app.json` lo creamos nosotros pero con cuidado de no cambiarlo luego, al ser un dominio inverso se sigue el patrón `com.usuario.app` en mi caso `com.svilla.mapsapp`. Para obtener la huella digital ejecutamos uno de los comandos de la derecha en función del SO que usemos, en mi caso Windows.

Windows

```
keytool -list -v -keystore $env:USERPROFILE\.android\debug.keystore -alias androiddebugkey -storepass android -keypass android
```

```
# Huella digital
C9:A1:25:FC:F2:88:50:C9:11:8D:2C:89:62:32:1E:C9:A2:05:FE:86
```

Agregar una aplicación para Android

Nombre del paquete *	com.svilla.mapsapp
Huella digital del certificado SHA-1 *	C9:A1:25:FC:F2:88:50:C9:11:8D:2C:89:62:32:1E:C9:A2:05:FE:86
Cancelar Listo	

14. Cuando damos a listo nos saldrá nuevas opciones que nos permitirán restringir o no las claves y la versión y cantidad de API que se deseen limitar:

Filtro Escribir el nombre o valor de la propiedad [?](#)

<input type="checkbox"/> Nombre del paquete	Huella digital	Editar
<input type="checkbox"/> com.svilla.mapsapp	C9:A1:25:FC:F2:88:50:C9:11:8D:2C:89:62:32:1E:C9:A2:05:FE:86	

Restricciones de API [?](#)

No restringir clave
Esta clave puede llamar a cualquier API

Restringir clave

32 API

API seleccionadas:

- Maps SDK for Android
- Directions API
- Distance Matrix API
- Maps Elevation API
- Maps Embed API
- Geocoding API
- Geolocation API
- Maps JavaScript API
- Roads API
- Maps SDK for iOS
- Time Zone API
- Places API
- Maps Static API
- Street View Static API
- Map Tiles API
- Routes API

15. Para generar una segunda clave para iOS hay que repetir todo el proceso. (OPCIONAL)
16. Ya hemos realizado los pasos 2 y 3 de la documentación (Crear y poner huella digital y crear API Key).
17. En la raíz creamos un fichero `.env` para introducir las credenciales con el código que nos proporciona la documentación.

```
EXPO_PUBLIC_GOOGLE_MAPS_API_KEY=AlzaSyAidPx2YBl-p-QzaHuqtJfUxetb1mAJc9c
```

18. En el fichero `app.json` siguiendo la estructura que nos indica la documentación ajustamos la estructura del objeto `android`

```
// Documentación
"android": {
  "config": {
    "googleMaps": {
```

```

        "apiKey": "process.env.GOOGLE_MAPS_API_KEY"
    },
},
}

// MI app.json
"android": {
    "package": "com.svilla.mapsapp",
    "config": {
        "googleMaps": {
            "apiKey": "AlzaSyAidPx2YBl-p-QzaHuqtJfUxetb1mAJs9c",
        }
    },
    "adaptiveIcon": {
        "backgroundColor": "#E6F4FE",
        "foregroundImage": "./assets/images/android-icon-foreground.png",
        "backgroundImage": "./assets/images/android-icon-background.png",
        "monochromelImage": "./assets/images/android-icon-monochrome.png"
    },
    "edgeToEdgeEnabled": true,
    "predictiveBackGestureEnabled": false
},

```

19. Antes de generar la compilación vamos a preparar el código para que gestione la API de Google cambiando el `import` del `MapView` y añadiendo su proveedor en el componente:

```

import MapView, { PROVIDER_GOOGLE } from 'react-native-maps'
<MapView style={styles.map}
provider={PROVIDER_GOOGLE}
showsMyLocationButton
showsUserLocation

initialRegion={{
    latitude: 39.459520762987665,
    longitude: -0.47044131140655987,
    latitudeDelta: 0,
    longitudeDelta: 0
}}
/>

```

20. Vamos a generar el `prebuild` con el comando `npx expo prebuild --clean` :

```

PS D:\RN\maps-app> npx expo prebuild --clean
env: load .env
env: export EXPO_PUBLIC_GOOGLE_MAPS_API_KEY
! Git branch has uncommitted file changes
> It's recommended to commit all changes before proceeding in case you want to revert generated changes.

? Continue with uncommitted changes? » (Y/n)

```

```

● PS D:\RN\maps-app> npx expo prebuild --clean
env: load .env
env: export EXPO_PUBLIC_GOOGLE_MAPS_API_KEY
! Git branch has uncommitted file changes
> It's recommended to commit all changes before proceeding in case you want to revert generated changes.

✓ Continue with uncommitted changes? ... yes

✓ Cleared android code
✓ Created native directory
✓ Updated package.json
✓ Finished prebuild
○ PS D:\RN\maps-app>

```

21. Continuamos con el `prebuild` de android con `npx expo run:android`:

```

○ PS D:\RN\maps-app> npx expo run:android
env: load .env
env: export EXPO_PUBLIC_GOOGLE_MAPS_API_KEY
> Building app...
Downloading https://services.gradle.org/distributions/gradle-8.14.3-bin.zip
.....10%.....20%.....30%.....40%.....50%
%.....60%.....70%.....80%.....90%.....100%

Welcome to Gradle 8.14.3!

Here are the highlights of this release:
- Java 24 support
- GraalVM Native Image toolchain selection
- Enhancements to test reporting
- Build Authoring improvements

For more details see https://docs.gradle.org/8.14.3/release-notes.html

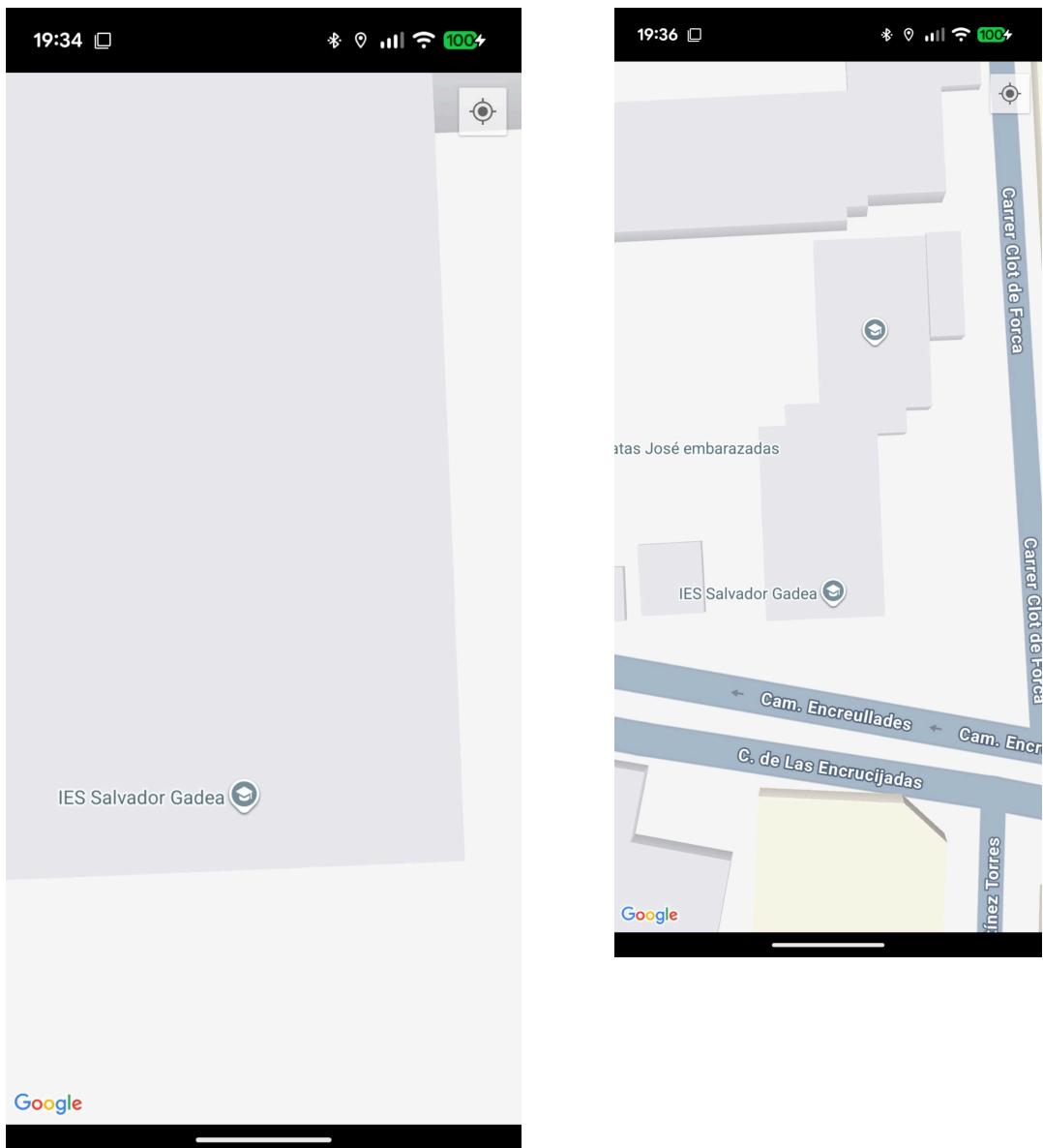
Starting a Gradle Daemon (subsequent builds will be faster)
Configuration on demand is an incubating feature.
<-----> 0% INITIALIZING [9s]
> Evaluating settings > Evaluating settings > Generating gradle-api-8.14.3.jar
□

BUILD SUCCESSFUL in 2m 13s
287 actionable tasks: 28 executed, 259 up-to-date
React Compiler enabled
Starting Metro Bundler

```

En caso de tener algún fallo en los porcentajes finales es posible que la máquina se haya quedado sin memoria RAM, quitar procesos y relanzar.

22. Tras relanzar la aplicación veremos algo como lo siguiente:



Se puede apreciar el icono de Google abajo a la izquierda y como se puede redimensionar el mapa, además el botón de centrar en la posición del usuario es funcional aunque en vuestro caso no se moverá mucho.

En caso de quitar los permisos y relanzar la aplicación veremos que nos pide el cambio de permisos y si se los damos nos cambia a la visualización del mapa.

5. Marcadores

Ya tenemos el mapa funcional, vamos a conocer sus utilidades a través de la documentación:

[GitHub - react-native-maps/react-native-maps: React Native Mapview component for iOS + Android](https://github.com/react-native-maps/react-native-maps)
React Native Mapview component for iOS + Android. Contribute to react-native-maps/react-native-maps development by creating an account on GitHub.

[react-native-maps/react-native-maps](https://github.com/react-native-maps/react-native-maps)

React Native Mapview component for iOS + Android

🔗 <https://github.com/react-native-maps/react-native-maps>

492 Contributors 204K Used by 258 Discussions 16k Stars

1. Una vez leída la documentación empezamos con el uso de marcadores.
2. Quitamos el autocierre del componente `MapView` y le ponemos otra etiqueta de cierre:

```

<MapView style={styles.map}
  provider={PROVIDER_GOOGLE}
  showsMyLocationButton
  showsUserLocation

initialRegion={{
  latitude: 39.459520762987665,
  longitude: -0.47044131140655987,
  latitudeDelta: 0,
  longitudeDelta: 0
}}
></MapView>

```

- Dentro del mapa creamos el componente `<Marker>` y ajustamos el import:

```

import MapView, { PROVIDER_GOOGLE, Marker } from 'react-native-maps'
<Marker
  coordinate={{
    latitude: 39.45919266033844,
    longitude: -0.46964300466007,
  }}
  title='Guimaraiz'
  description='Lugar de buenos almuerzos'
/>

```

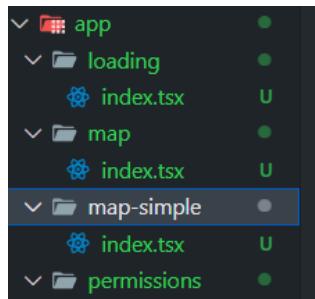


```

<Marker
  coordinate={{
    latitude: 39.45919266033844,
    longitude: -0.46964300466007,
  }}
  title='Guimaraiz'
  description='Lugar de buenos almuerzos'
  on? El tipo '{ coordinate: { latitude: number; longitude: number; } }' tiene los siguientes métodos de acción:
  /> ⚡ onAccessibilityAction? (property) onAccessibilityAction?: ...
  /MapV ⚡ onAccessibilityEscape?
  aFeAr ⚡ onAccessibilityTap?
  ⚡ onBlur?
  ⚡ onCalloutPress?
  ⚡ onDeselect?
  ⚡ onDrag?
  ⚡ onDragEnd?
  ⚡ onDragStart?
  ⚡ onFocus?
  ⚡ onLayout?
  ⚡ onMagicTap?

```

- Todos los marcadores en su componente tienen acceso a los métodos de acción que pueden ser sobreescritos para darle la funcionalidad deseada.
- Para finalizar vamos a copiar la carpeta que contiene nuestra pantalla de `map` y a la copia le daremos el nombre de `map-simple` para tener una versión sencilla del mapa.



6. Eliminamos los marcadores del `map` original y lo volvemos autocontenido para poder trabajar con él.

6. Acciones para seguir y ver la ubicación del usuario

En esta sección vamos a ver y gestionar la ubicación del dispositivo en movimiento en tiempo real.

1. Creamos el fichero `core/actions/location/location.ts`, esta carpeta y fichero tendrán otra acción que se generará en nuestra aplicación.
2. Preparamos una función en nuestra nueva y pequeña librería llamada `getCurrent` :

```
import * as Location from 'expo-location'

export const getCurrent = async() => {
  //TODO
}
```

3. Creamos en la carpeta de `infrastructure/interfaces` un fichero llamado `lat-lng.ts` :

```
export interface LatLng{
  latitude: number;
  longitude: number;
}
```

4. Modificamos la función anterior `getCurrent` :

```
export const getCurrent = async():Promise<LatLng> => {
  try{
    const { coords } = await Location.getCurrentPositionAsync({
      accuracy: Location.Accuracy.Highest
    });

    return {
      latitude: coords.latitude,
      longitude: coords.longitude
    }
  }catch (error){
    throw new Error('Error obteniendo la localización del usuario');
  }
}
```



⚠ Esta función es la que va a hacer peticiones a la librería `Location` que hará las comunicaciones con el API de Maps donde recibiremos un objeto `coords` que tendrá las coordenadas en alta precisión del usuario y que nosotros usaremos para devolver nuestro objeto modelo de coordenadas.

5. Creamos la siguiente función llamada `watchCurrentPosition` :

```

export const watchCurrentPosition = (
  locationCallback: (location: LatLng) => void
) => {

  return Location.watchPositionAsync({
    accuracy: Location.Accuracy.Highest,
    timeInterval: 1000, //milisegundos
    distanceInterval: 10 //metros
  }, ({coords}) =>{
    locationCallback({
      latitude: coords.latitude,
      longitude: coords.longitude
    })
  })
}

```

⚠️ La siguiente función es muy compleja de explicar y de entender en profundidad así que vamos a intentar hacerlo:

1. Cuando la función `watchCurrentPosition` sea llamada recibirá como parámetro una función `callback` con un parámetro de entrada un objeto de tipo `LatLng` y **en este momento** devolverá un `void` y no hará nada.
2. La función devuelve el resultado de la función `watchPositionAsync` que será una subscripción con el único propósito de recibirla y usarla para cancelar la escucha al GPS.
3. La función `watchPositionAsync` recibe un objeto configuración y una función que usará como `callback` cada vez que el GPS genere una posición.
4. La función `watchCurrentPosition` se ejecuta una única vez pero la función `locationCallback` se ejecuta n veces, tantas como el GPS genere una nueva posición y avise al SO para que se lo notifique a Expo Go que está atento mediante la subscripción.
5. El GPS sigue activo y emitiendo periódicamente porque tiene asociado un subscriptor que le ha dado un objeto de configuración periódica y hasta que dicho subscriptor no cese su actividad no parará de emitir.
6. La función `locationCallback` es llamada por cada nueva generación de una posición y esta posición `{coords}` es transformada en un objeto `LatLng`.
7. El cuerpo del `locationCallback` **actualmente** no procesa la petición pero se cambiará para que genere un cambio que alguien escuchará y al que reaccionará. Por ejemplo `(location: LatLng) => setLocation(location);`
8. El subscriptor lo recibe un gestor externo que recibe el objeto y tendrá la capacidad de eliminarlo o destruirlo.

6. Creamos un nuevo fichero en la carpeta `presentation/store/useLocation.ts` y refactorizamos el nombre del fichero `usePermissions` a `usePermissionsStore`, al recomilar nos dará fallo y nos pedirá cambiar las dependencias, decimos que sí y nos aseguramos de que estén cambiadas y compile la aplicación sin problemas.
7. Hacemos la implementación de `useLocationStore.ts` :

```

import { getCurrentLocation, watch currentPosition } from '@/core/actions/location/location';
import { LatLng } from '@/infrastructure/interfaces/lat-lng';
import { LocationSubscription } from 'expo-location';
import { create } from 'zustand';

interface LocationState {

```

```

lastKnownLocation: LatLng | null;
//lastKnownLocation?: LatLng;
userLocationList: LatLng[];
watchSubscriptionID: LocationSubscription | null;
//watchSubscription?: LocationSubscription;

getLocation: () => Promise<LatLng>;
watchLocation: () => void;
clearWatchLocation: () => void;
}

export const useLocationStore = create<LocationState>()((set, get) => ({
lastKnownLocation: null,
userLocationList: [],
watchSubscriptionID: null,

getLocation: async() => {
  const location = await getCurrentLocation();
  set({ lastKnownLocation: location })
  return location
},
watchLocation: async() =>{
  const oldSubscription = get().watchSubscriptionID;
  if (oldSubscription !== null){
    get().clearWatchLocation();
  }
}

const watchSubscription = await watchCurrentPosition(
  (latLng) => {
    set({
      lastKnownLocation: latLng,
      userLocationList: [...get().userLocationList, latLng],
    });
  }

  //Forma declarativa sin get
  //set((state) => ({
  //lastKnownLocation: latLng,
  //userLocationList: [...state.userLocationList, latLng],
  //)));
}

set({watchSubscriptionID: watchSubscription});
),

clearWatchLocation: () => {
  const subscription = get().watchSubscriptionID;
  if(subscription !== null){
    subscription.remove();
    set({ watchSubscriptionID: null });
  }
}

)));

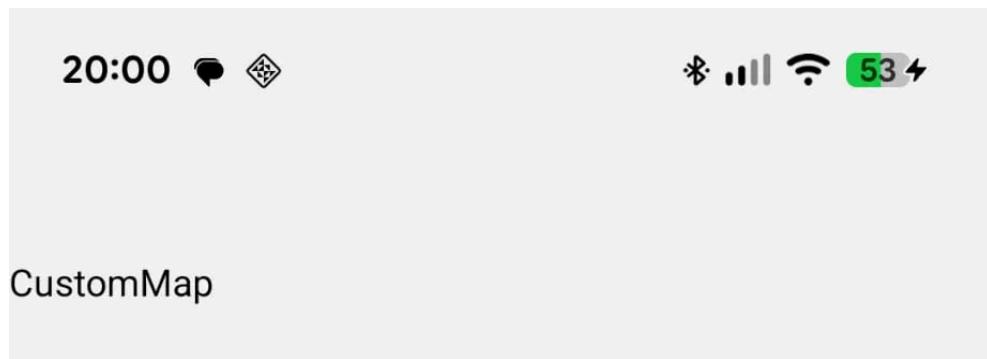
```

- En estos momentos ya tenemos toda la lógica de negocio preparada para la gestión de un mapa que va a seguir a un usuario en tiempo real ya sea una vez de forma puntual o a través de un envío continuo de pulsaciones del GPS para seguir el movimiento.

7. Creación de un **CustomMap**

Vamos a crear nuestro componente mapa para adecuarlo a nuestras necesidades.

- Creamos en `shared` nuestro fichero `map/CustomMap.tsx`.
- Creamos el componente con `rnfe` y ponemos los `SafeAreaView`.
- Sustituimos el componente mapa **FUNCIONAL** por nuestro componente para ver si se efectúa el cambio. Debemos ver el cambio sin problemas siempre y cuando tengamos permisos.



- Continuamos preparando la personalización del componente `CustomMap` incluyendo el envoltorio de la `SafeAreaView` para que sea un componente flexible.

```
import { LatLng } from '@/infrastructure/interfaces/lat-lng';
import { Text } from 'react-native'
import { SafeAreaView, SafeAreaViewProps } from 'react-native-safe-area-context';

interface Props extends SafeAreaViewProps{
  showUserLocation?: boolean;
  initialLocation?: LatLng;
}

const CustomMap = ({showUserLocation = true, initialLocation, ...rest}: Props) => {
  return (
    <SafeAreaView {...rest}>
      <Text>CustomMap</Text>
    </SafeAreaView>
  )
}
export default CustomMap
```

- Modificamos la entrada de datos del nuevo componente en la pantalla `MapScreen`.

```
<CustomMap
  initialLocation={{
    latitude: 39.459520762987665,
    longitude: -0.47044131140655987,
  }}>
```

```
    showUserLocation  
  />
```

6. Cortamos el `MapView` anterior y lo pegamos en nuestro componente personalizado **junto a los estilos** que tiene asociado el `MapView`. Volvemos a importar las clases adecuadas para que no hayan errores y probamos que nos vuelve a funcionar el mapa. No borramos los `styles` de la pantalla del mapa, solo borramos el estilo `map`.

```
import { StyleSheet } from 'react-native'  
import { LatLng } from '@/infrastructure/interfaces/lat-lng';  
import MapView, { PROVIDER_GOOGLE, Marker } from 'react-native-maps'  
import { SafeAreaView, SafeAreaViewProps } from 'react-native-safe-area-context';  
  
interface Props extends SafeAreaViewProps{  
  showUserLocation?: boolean;  
  initialLocation?: LatLng;  
}  
  
const CustomMap = ({showUserLocation = true, initialLocation, ...rest}: Props) => {  
  return (  
    <SafeAreaView {...rest}>  
      <MapView style={styles.map}  
        provider={PROVIDER_GOOGLE}  
        showsMyLocationButton  
        showsUserLocation  
  
        initialRegion={{  
          latitude: 39.459520762987665,  
          longitude: -0.47044131140655987,  
          latitudeDelta: 0,  
          longitudeDelta: 0  
        }}  
      />  
    </SafeAreaView>  
  )  
}  
export default CustomMap;  
  
const styles = StyleSheet.create({  
  container: {  
    flex: 1  
  },  
  map:{  
    width: '100%',  
    height: '100%',  
    //backgroundColor: 'red'  
  }  
});
```

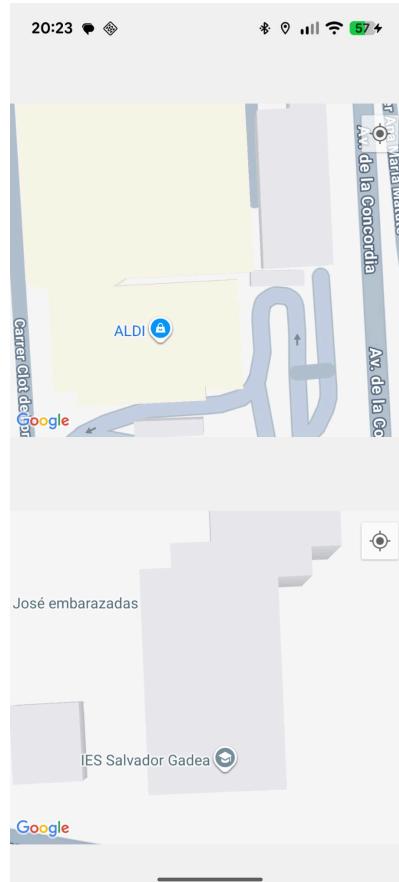
7. Una de las ventajas de permitir añadir estilos de esta forma es que si queremos podemos crear 2 mapas en la misma pantalla:

```
<CustomMap  
  initialLocation={{  
    latitude: 39.459520762987665,  
    longitude: -0.47044131140655987,
```

```

        }}
        style={{height: '50%'}}
    />
<CustomMap
    initialLocation={{
        latitude: 39.459520762987665,
        longitude: -0.47044131140655987,
    }}
    style={{height: '50%'}}
/>

```



8. Modificamos un poco nuestro componente para ajustarlo a la entrada de datos del componente:

```

import { StyleSheet } from 'react-native'
import { LatLng } from '@@/infrastructure/interfaces/lat-lng';
import MapView, { PROVIDER_GOOGLE, Marker } from 'react-native-maps'
import { SafeAreaView, SafeAreaViewProps } from 'react-native-safe-area-context';

interface Props extends SafeAreaViewProps{
    showUserLocation?: boolean;
    initialLocation?: LatLng;
}

const CustomMap = ({showUserLocation = true, initialLocation, ...rest}: Props) => {
    if(initialLocation === undefined){
        initialLocation = {
            latitude: 39.459520762987665,
            longitude: -0.47044131140655987
        }
    }
}

```

```

        }

    }

    return (
      <SafeAreaView {...rest}>
        <MapView style={styles.map}
          provider={PROVIDER_GOOGLE}
          showsMyLocationButton
          showsUserLocation = {showUserLocation}

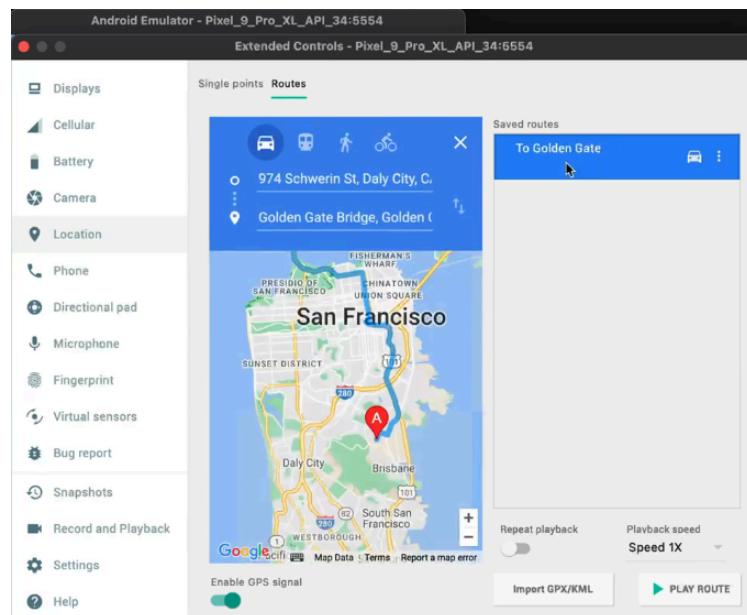
          initialRegion={{
            latitude: initialLocation.latitude,
            longitude: initialLocation.longitude,
            latitudeDelta: 0,
            longitudeDelta: 0
          }}
        />
      </SafeAreaView>
    )
  }
}

export default CustomMap;

const styles = StyleSheet.create({
  container: {
    flex: 1
  },
  map:{ width: '100%', height: '100%', //backgroundColor: 'red' }
});

```

9. A través de la configuración del simulador dando a la opción de ... → Localización → Rutas → Crear una ruta y simularla:



10. Con el simulador veremos que nuestra ubicación se actualiza.

8. Ubicación del usuario y seguir movimiento

Estamos listos para gestionar 1 ubicación ya sea la que está por defecto o la posición del propio usuario pero no podemos gestionar aún una pulsación continua de posiciones ene movimiento de forma automática, así que vamos a ello.

1. Vamos a nuestra pantalla de mapa y vamos a modificar el código:

```
import { ActivityIndicator, StyleSheet } from 'react-native'
import { SafeAreaView } from 'react-native-safe-area-context';
import CustomMap from '@/presentation/components/shared/maps/CustomMap';
import { useLocationStore } from '@/presentation/store/useLocationStore';
import { useEffect } from 'react';

const MapScreen = () => {

  const {lastKnownLocation, getLocation} = useLocationStore();

  useEffect(() => {
    if(lastKnownLocation === null){
      getLocation();
    }
  }, []);

  //return de prueba fuera del if, se borra tras probarlo

  if(lastKnownLocation === null){
    return <SafeAreaView style={{flex: 1, justifyContent: 'center', alignItems: 'center'}}>
      <ActivityIndicator />
    </SafeAreaView>
  }

  return (
    <SafeAreaView style={styles.container}>
      <CustomMap
        initialLocation={{
          latitude: 39.459520762987665,
          longitude: -0.47044131140655987,
        }}
        style={{height: '100%'}}
      />
    </SafeAreaView>
  )
}

export default MapScreen;

const styles = StyleSheet.create({
  container: {
    flex: 1
  }
});
```

2. Al recargar la aplicación veremos que durante un instante nos saldrá un indicador de que se debe recargar el contexto para obtener la ubicación y acto seguido se mostrará el mapa dado que esa recarga se hará de forma automática siempre que no haya fallado nada en el código.



3. Cambiamos el código en la pantalla del mapa para enviar nuestra ubicación en lugar de una que hemos decidido arbitrariamente:

```
// ANTES
initialLocation={{{
  latitude: 39.459520762987665,
  longitude: -0.47044131140655987,
}}}

// AHORA
initialLocation={lastKnownLocation}
```

4. Con el cambio vemos que efectivamente se marca el lugar donde estamos.

5. Vamos a preparar el mapa para que soporte el seguimiento continuo en el componente personalizado:

```
import { StyleSheet } from 'react-native'
import { LatLng } from '@/infrastructure/interfaces/lat-lng';
import MapView, { PROVIDER_GOOGLE, Marker } from 'react-native-maps'
import { SafeAreaView, SafeAreaViewProps } from 'react-native-safe-area-context';
import { useLocationStore } from '@/presentation/store/useLocationStore';
import { useEffect } from 'react';

interface Props extends SafeAreaViewProps{
  showUserLocation?: boolean;
  initialLocation?: LatLng;
}

const CustomMap = ({showUserLocation = true, initialLocation, ...rest}: Props) => {
  if(initialLocation === undefined){
    initialLocation = {
      latitude: 39.459520762987665,
      longitude: -0.47044131140655987
    }
  }

  const {watchLocation, clearWatchLocation} = useLocationStore();
```

```

useEffect(() => {
  watchLocation();
  return () => {
    clearWatchLocation();
  }
}, [])

return (
  <SafeAreaView {...rest}>
    <MapView style={styles.map}
      provider={PROVIDER_GOOGLE}
      showsMyLocationButton
      showsUserLocation = {showUserLocation}

      initialRegion={{
        latitude: initialLocation.latitude,
        longitude: initialLocation.longitude,
        latitudeDelta: 0,
        longitudeDelta: 0
      }}
    />
  </SafeAreaView>
)
}
export default CustomMap;

```

```
// USELOCATIONSTORE
```

```

LOG []
LOG [{"latitude": 39.4923073, "longitude": -0.4615172}]
LOG [{"latitude": 39.4923073, "longitude": -0.4615172}, {"latitude": 39.492397, "longitude": -0.4615361}]

LOG []
LOG [{"latitude": 39.4923073, "longitude": -0.4615172}]
LOG [{"latitude": 39.4923073, "longitude": -0.4615172}, {"latitude": 39.492397, "longitude": -0.4615361}, {"latitude": 39.4923196, "longitude": -0.4614726}]
LOG [{"latitude": 39.4923073, "longitude": -0.4615172}, {"latitude": 39.492397, "longitude": -0.4615361}, {"latitude": 39.4923196, "longitude": -0.4614726}, {"latitude": 39.4922781, "longitude": -0.461586}]
LOG [{"latitude": 39.4923073, "longitude": -0.4615172}, {"latitude": 39.492397, "longitude": -0.4615361}, {"latitude": 39.4923196, "longitude": -0.4614726}, {"latitude": 39.4922781, "longitude": -0.461586}, {"latitude": 39.4923399, "longitude": -0.4614984}]

```

Aquí ya hemos preparado el código para recibir las actualizaciones de la posición del usuario a lo largo del tiempo y nuestra posición se mueve pero nuestro punto no tiene el foco de la visualización ni hay seguimiento por parte de la pantalla.

9. Mover la cámara para seguir al usuario

Vamos a implementar el seguimiento de la pantalla al punto que se actualiza:

1. En el componente `CustomMap` actualizamos el código creando una referencia al mapa, añadiendo esa referencia al mapa y creando una función que va a gestionar el centrado dinámico de la cámara cada vez que hay un cambio de posición:

```

import { StyleSheet } from 'react-native'
import { LatLng } from '@/infrastructure/interfaces/lat-lng';
import MapView, { PROVIDER_GOOGLE, Marker } from 'react-native-maps'

```

```

import { SafeAreaView, SafeAreaViewProps } from 'react-native-safe-area-context';
import { useLocationStore } from '@/presentation/store/useLocationStore';
import { useEffect, useRef } from 'react';

interface Props extends SafeAreaViewProps{
  showUserLocation?: boolean;
  initialLocation?: LatLng;
}

const CustomMap = ({showUserLocation = true, initialLocation, ...rest}: Props) => {
  if(initialLocation === undefined){
    initialLocation = {
      latitude: 39.459520762987665,
      longitude: -0.47044131140655987
    }
  }

  const mapRef = useRef<MapView>(null);
  const {watchLocation, clearWatchLocation} = useLocationStore();

  useEffect(() => {
    watchLocation();
    return () => {
      clearWatchLocation();
    }
  }, [])

  const moveCameraToLocation = (latLng: LatLng) => {
    if(!mapRef.current) return;
    mapRef.current.animateCamera({
      center: latLng
    })
  }

  return (
    <SafeAreaView {...rest}>
      <MapView
        ref={mapRef}
        style={styles.map}
        provider={PROVIDER_GOOGLE}
        showsMyLocationButton
        showsUserLocation = {showUserLocation}

        initialRegion={{
          latitude: initialLocation.latitude,
          longitude: initialLocation.longitude,
          latitudeDelta: 0,
          longitudeDelta: 0
        }}
      />
    </SafeAreaView>
  )
}

export default CustomMap;

```

```
const styles = StyleSheet.create({
  container: {
    flex: 1
  },
  map:{ 
    width: '100%', 
    height: '100%', 
    //backgroundColor: 'red' 
  }
});
```



A través del `useEffect` crea la funcionalidad para actualizar la cámara. Esto creará una situación en la que no podremos perder el foco de nuestra posición en movimiento pero se solucionará.

2. La solución a la tarea para por recibir el atributo `lastKnownLocation` y hacer un `useEffect` que escuche sus cambios comprobando si es nulo para llamar a la actualización de la cámara:

```
// Cabecera
const {watchLocation, clearWatchLocation, lastKnownLocation} = useLocationStore();

...
useEffect(() => {
  if(lastKnownLocation){
    moveCameraToLocation(lastKnownLocation);
  }
}, [lastKnownLocation])
```

3. Con esto podemos pasar a seguir haciendo crecer la aplicación.

10. FAB para personalizar acciones

Vamos a crear un `Floating Action Button` para personalizar las acciones de nuestro mapa:

1. En `shared` creamos un nuevo componente:

```
import { Text } from 'react-native';
import { SafeAreaView } from 'react-native-safe-area-context';

interface Props{

}

const FAB = ({}: Props) => {
  return (
    <SafeAreaView>
      <Text>FAB</Text>
    </SafeAreaView>
  )
}
export default FAB
```

2. En la vista del mapa, dentro de la vista y debajo del mapa creamos el componente FAB y comprobamos que todo funciona al recargar la app.

3. Creamos los estilos del `FAB` para poder visualizarlo:

```
const styles = StyleSheet.create({
  btn: {
    zIndex: 99,
    position: 'absolute',
    height: 50,
    width: 50,
    borderRadius: 30,
    backgroundColor: 'black',
    justifyContent: 'center',
    alignItems: 'center',
    shadowOpacity: 0.3,
    shadowOffset: {
      height: 0.27,
      width: 4.5,
    },
    elevation: 5,
  }
});
```

4. Los `Props` podemos extenderlos desde `View` o desde `Pressable`, en nuestro caso no extendemos de ninguna para gestionar manualmente lo que nos llega, pero ambas soluciones son válidas dependiendo de la funcionalidad que se espere de dicho componente.

```
import { Ionicons } from '@expo/vector-icons';
import { StyleSheet, Pressable, ViewStyle, StyleProp } from 'react-native'

interface Props{
  onPress: () => void;
  style?: StyleProp<ViewStyle>;
}

const FAB = ({onPress, style}: Props) => {
  return (
    <Pressable style={[styles.btn, style]} onPress={onPress}>
      <Ionicons name="add-circle"/>
    </Pressable>
  )
}
export default FAB;

const styles = StyleSheet.create({
  btn: {
    zIndex: 99,
    position: 'absolute',
    height: 50,
    width: 50,
    borderRadius: 30,
    backgroundColor: 'black',
    justifyContent: 'center',
    alignItems: 'center',
    shadowOpacity: 0.3,
    shadowOffset: {
      height: 0.27,
```

```
        width: 4.5,  
    },  
    elevation: 5,  
}  
});
```

5. Ahora se quejará nuestro componente `FAB` en la vista dado que le faltan datos de entrada por entregar:

```
style={{  
    bottom: 70,  
    right: 20,  
}}
```

6. Continuamos personalizando el nombre del ícono que vamos a mostrar:

```
interface Props{  
    onPress: () => void;  
    style?: StyleProp<ViewStyle>;  
    iconName: keyof typeof Ionicons.glyphMap  
}  
  
...  
  
<Ionicons name={iconName} color="white" size={35}/>
```

7. Vamos a la vista y le damos nombre al ícono que tiene que recibir:

```
<FAB  
    iconName="add"  
    onPress={() => {}}  
    style={{  
        bottom: 20,  
        right: 20,  
    }}  
/>
```

8. Ya tenemos preparada la acción personalizada del mapa.

11. Acciones con el mapa

Vamos a implementar la funcionalidad personalizada.

1. Vamos a hacer que cuando el usuario quite el foco de la actualización automática respecto el movimiento del usuario se use como indicador de que queremos dejar de seguir al usuario y hacer búsqueda libre. Vamos a cambiar cosa en el componente personalizado del mapa.
Creamos el estado que va a gestionar ese seguimiento y actualizamos la dependencia de reacción de la cámara.

```
const mapRef = useRef<MapView>(null);  
const [isFollowingUser, setIsFollowingUser] = useState(true);  
  
...  
  
useEffect(() => {  
    if(lastKnownLocation && isFollowingUser){  
        moveCameraToLocation(lastKnownLocation);
```

```
        }, [lastKnownLocation, isFollowingUser])
```

2. Vamos a generar la situación de desactivación de la variable cuando movemos el mapa:

```
<MapView  
ref={mapRef}  
onTouchStart={() => setIsFollowingUser(false)}
```

3. Vamos cambiar la funcionalidad del botón para gestionar la localización del usuario:

```
// 1º → Asignamos función que vamos a crear  
onPress={moveToCurrentLocation}  
  
// 2º → Creamos función e importamos getLocation()  
const {getLocation, watchLocation, clearWatchLocation, lastKnownLocation} = useLocationStore();  
  
const moveToCurrentLocation = async() => {  
  if(!lastKnownLocation){  
    moveCameraToLocation(initialLocation);  
  }else{  
    moveCameraToLocation(lastKnownLocation);  
  }  
  
  const location = await getLocation();  
  if(!location) return;  
  
  moveCameraToLocation(location);  
}
```

4. Comprobamos que funciona la vuelta de la cámara al usuario.

5. Creamos un segundo botón que hará la gestión de vuelta al foco del usuario:

```
<FAB  
iconName={ isFollowingUser ? 'walk-outline' : 'accessibility-outline' }  
onPress={() => setIsFollowingUser(!isFollowingUser)}  
style={{  
  bottom: 80,  
  right: 20,  
}}  
/>
```

6. Ya tenemos una funcionalidad de seguimiento y control de cámara.

12. Polylines

Vamos a marcar el camino que ha seguido el usuario durante su activación continua del GPS, de ello se van a encargar el concepto de **Polyline**.

1. Importamos la variable `userLocationList` y convertimos la etiqueta del `MapView` de autocerrada a doble etiqueta y añadimos la etiqueta `Polyline`:

```

const {watchLocation, clearWatchLocation, lastKnownLocation, getLocation, userLocationList} = useLocationStore();

...

<MapView
  ref={mapRef}
  onTouchStart={() => setIsFollowingUser(false)}
  style={styles.map}
  provider={PROVIDER_GOOGLE}
  showsMyLocationButton
  showsUserLocation = {showUserLocation}

  initialRegion={{
    latitude: initialLocation.latitude,
    longitude: initialLocation.longitude,
    latitudeDelta: 0,
    longitudeDelta: 0
  }}
>
  <Polyline coordinates={userLocationList}/>
</MapView>

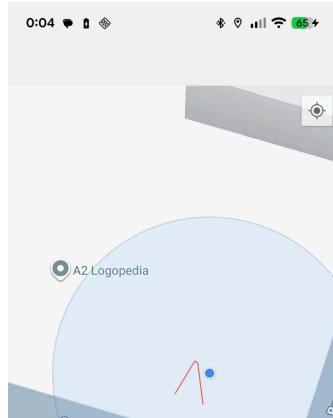
```

2. Con esta etiqueta se mostrará el camino que el usuario ha ido recorriendo, aquel que este grabado en nuestra lista de posiciones. Para personalizar la `Polyline` hay muchas propiedades, nosotros usaremos la propiedad `strokeColor`.

```

<Polyline
  coordinates={userLocationList}
  strokeColor={'red'}
/>

```



Crea otro botón que genere el comportamiento para mostrar y ocultar el `Polyline` según el usuario quiera

3. Creamos la funcionalidad:

```
const [isShowingPolyline, setIsShowingPolyline] = useState(true);
```

```

...
<FAB
  iconName={ isShowingPolyline ? 'eye-outline' : 'eye-off-outline'}
  onPress={() => setIsShowingPolyline(!isShowingPolyline)}
  style={{
    bottom: 140,
    right: 20,
  }}
/>

```

4. Con esto finalizamos la sección de Maps

13. Uso de **AsyncStorage** para datos no confidenciales

Vamos a gestionar la ubicación del usuario cada vez que se actualiza, la vamos a hacer persistente incluso ante el cierre de la aplicación y tras su reinicio posterior vamos a recuperar dicha ubicación en persistencia.

1. Creamos dentro de `core` el fichero `persistence/locationStorage.ts` .

```

// core/persistence/locationStorage.ts
import AsyncStorage from '@react-native-async-storage/async-storage';
import { LatLng } from '@/infrastructure/interfaces/lat-lng';

const LAST_LOCATION_KEY = 'LAST_KNOWN_LOCATION';

/**
 * Guarda la última localización del usuario en almacenamiento local
 */
export const saveLastLocation = async (location: LatLng): Promise<void> => {
  try {
    await AsyncStorage.setItem(
      LAST_LOCATION_KEY,
      JSON.stringify(location)
    );
  } catch (error) {
    console.error('Error guardando la localización', error);
  }
};

/**
 * Recupera la última localización guardada (si existe)
 */
export const getLastLocation = async (): Promise<LatLng | null> => {
  try {
    const value = await AsyncStorage.getItem(LAST_LOCATION_KEY);
    return value ? JSON.parse(value) as LatLng : null;
  } catch (error) {
    console.error('Error recuperando la localización', error);
    return null;
  }
};

/**
 * Elimina la localización guardada
 */

```

```

export const clearLastLocation = async (): Promise<void> => {
  try {
    await AsyncStorage.removeItem(LAST_LOCATION_KEY);
  } catch (error) {
    console.error('Error limpiando la localización', error);
  }
};

```

2. Vamos a nuestro `useLocationStore.ts` y añadimos las importaciones para gestionar el uso de `AsyncStorage` y creamos una nueva función para recuperar la última ubicación:

```

import { saveLastLocation, getLastLocation } from '@core/persistence/locationStorage';

...

// Definimos la nueva función en la interfaz
loadLastStoredLocation: () => Promise<void>;
...

// Creamos la función que hemos definido en la interfaz
loadLastStoredLocation: async () => {
  const storedLocation = await getLastLocation();
  if (storedLocation) {
    set({ lastKnownLocation: storedLocation });
  }
};

...

// Actualizamos la función watchCurrentPosition
const subscription = await watchCurrentPosition((latLng) => {
  set((state) => ({
    lastKnownLocation: latLng,
    userLocationList: [...state.userLocationList, latLng],
  }));
}

// Persistimos SOLO la última
saveLastLocation(latLng);
});

...

// En nuestro Provider incorporamos la persistencia
const { loadLastStoredLocation } = useLocationStore();

useEffect(() => {
  loadLastStoredLocation();
}, []);

```

3. Gracias a estos cambios ahora Zustand puede guardar, a través de la librería que hemos creado, los últimos datos de forma persistente y cuando se reinicia la aplicación se cargará sobre la variable `lastKnownLocation` el último valor de coordenadas que haya en memoria para que luego sea el GPS el que sobreescriba esa ubicación.