GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2021-2022

UNIVERSIDAD DE GRANADA

PERIFÉRICOS Y DISPOSITIVOS DE INTERFAZ HUMANA

QR READER

Alejandro Sánhez Hens
Paula Molina Castillo

1. *Introducción*

El siguiente proyecto realizado para la asignatura de "*Periféricos y Dispositivos de Interfaz Humana*" consiste en una aplicación móvil, compatible tanto para Android como para IOS, mediante la cual podremos escanear **códigos QR** relacionados a enlaces de Internet y **posiciones geográficas** de Google Maps.

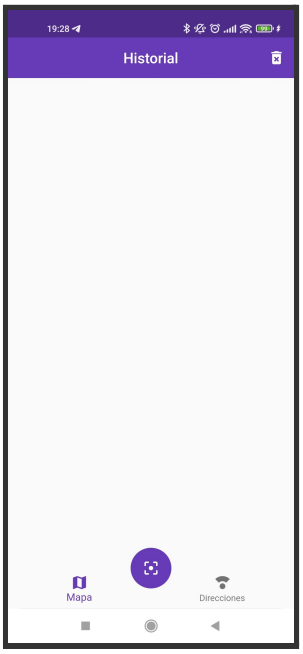
El objetivo de este proyecto es la creación de una aplicación fácil de usar y útil. Durante todo el proceso de desarrollo hemos intentado crear una interfaz que sea lo más accesible posible, intuitiva y atendiendo a unos ciertos criterios de calidad, para que así el usuario pueda disfrutar de una buena experiencia durante el uso de nuestra aplicación.

2. Explicación del funcionamiento de la aplicación

Esta aplicación, como bien hemos mencionado anteriormente, consiste en un escáner de códigos QR. Gracias a este escáner podremos escanear:

- Códigos QR que nos lleven a enlaces de páginas web de Internet.
- Códigos QR que nos lleven a ubicaciones de Google Maps.

En la pantalla principal de la aplicación encontraremos una pantalla, inicialmente, en blanco.

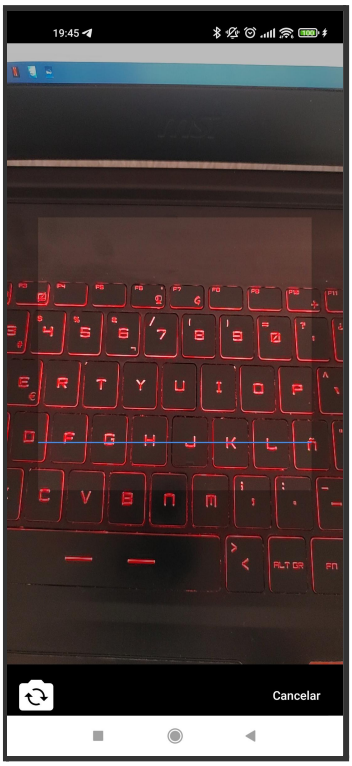


Como podemos ver en la imágen anterior, nuestra aplicación nos muestra primeramente un historial en blanco, dicho historial corresponde a nuestros últimos escaneos. En este historial se nos mostrarán todos los escaneos realizados hasta la fecha. Dichos escaneos están catalogados entre escaneos de Google Maps o escaneos de direcciones a páginas web de Internet.

En la parte superior derecha de la pantalla nos encontramos con un botón con un icono de un cubo de basura. Este botón nos permitirá borrar todos los registros de escaneos realizados.

A continuación, si miramos la parte inferior de la pantalla podemos ver 3 botones.

El botón central es el botón que nos permitirá acceder al escáner. Si pulsamos en él, automáticamente accedemos a la pantalla del escáner y mediante el cual podremos hacer el escaneo de códigos QR.



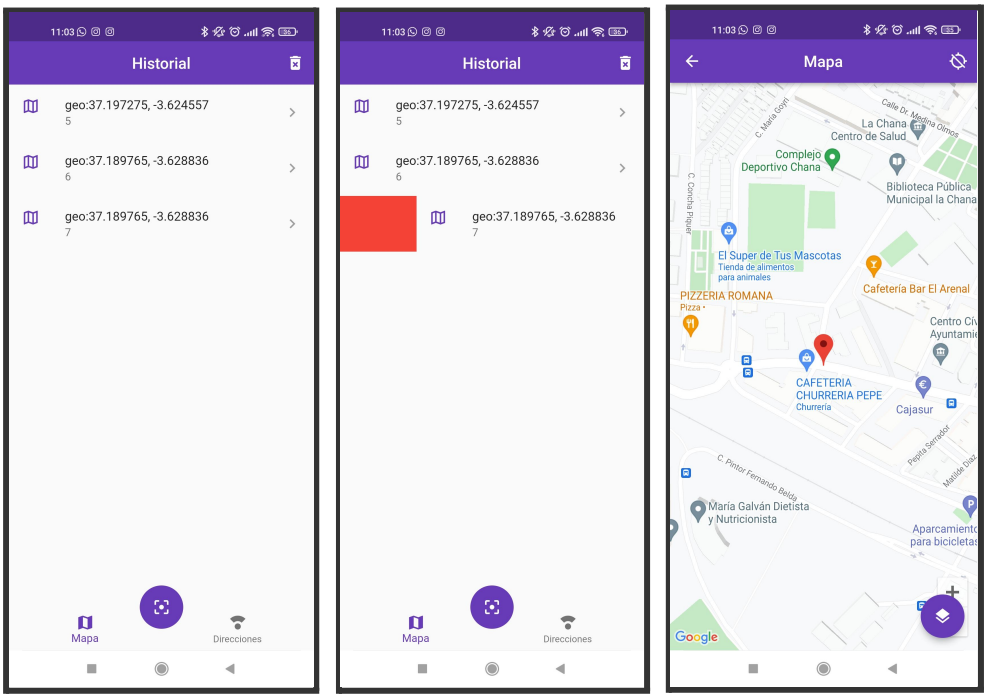
Como podemos ver en esta pantalla, la aplicación ha accedido a la cámara del dispositivo y nos muestra una zona de escaneo, y dos botones en la parte inferior de la pantalla. Dichos botones nos permiten permutar entre la cámara frontal y trasera del dispositivo, o cancelar el escaneo.

Al escanear un código QR accederemos a la información de dicho código y se guardará en su sección correspondiente. Dichas secciones son accesibles a través de los dos botones de la pantalla principal, situados a ambos lados del botón central.

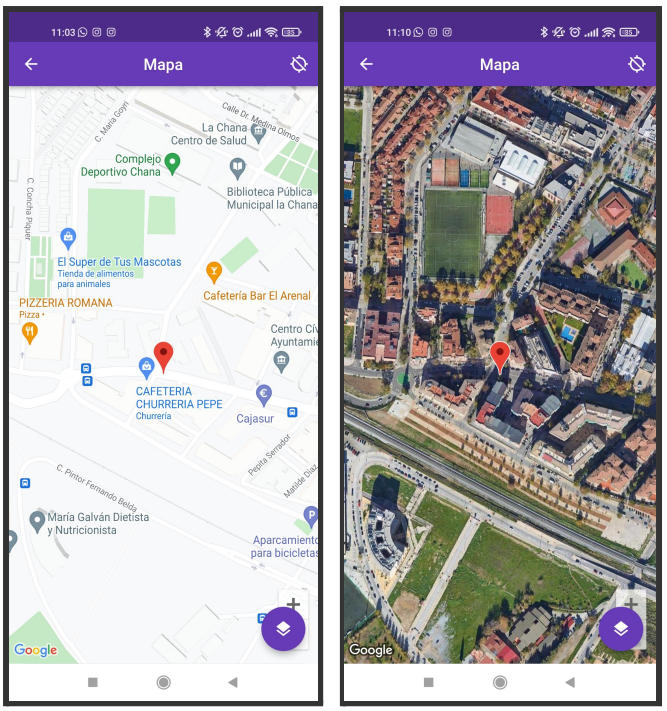
Comenzamos analizando la sección correspondiente a los escaneos de ubicaciones de Google Maps. Esta sección es accesible gracias al botón situado en la parte inferior izquierda de la pantalla, el cual nos muestra un icono de un mapa y un texto debajo del mismo, el cual pone “Mapa”.

Si accedemos a la sección de “Mapa” de la aplicación nos encontramos con un historial de las ubicaciones escaneadas hasta la fecha.

A parte de mostrarlas también podemos volver a acceder a ellas, pulsando en la ubicación que queramos, o eliminar una ubicación determinada, deslizando dicha ubicación de izquierda a derecha.

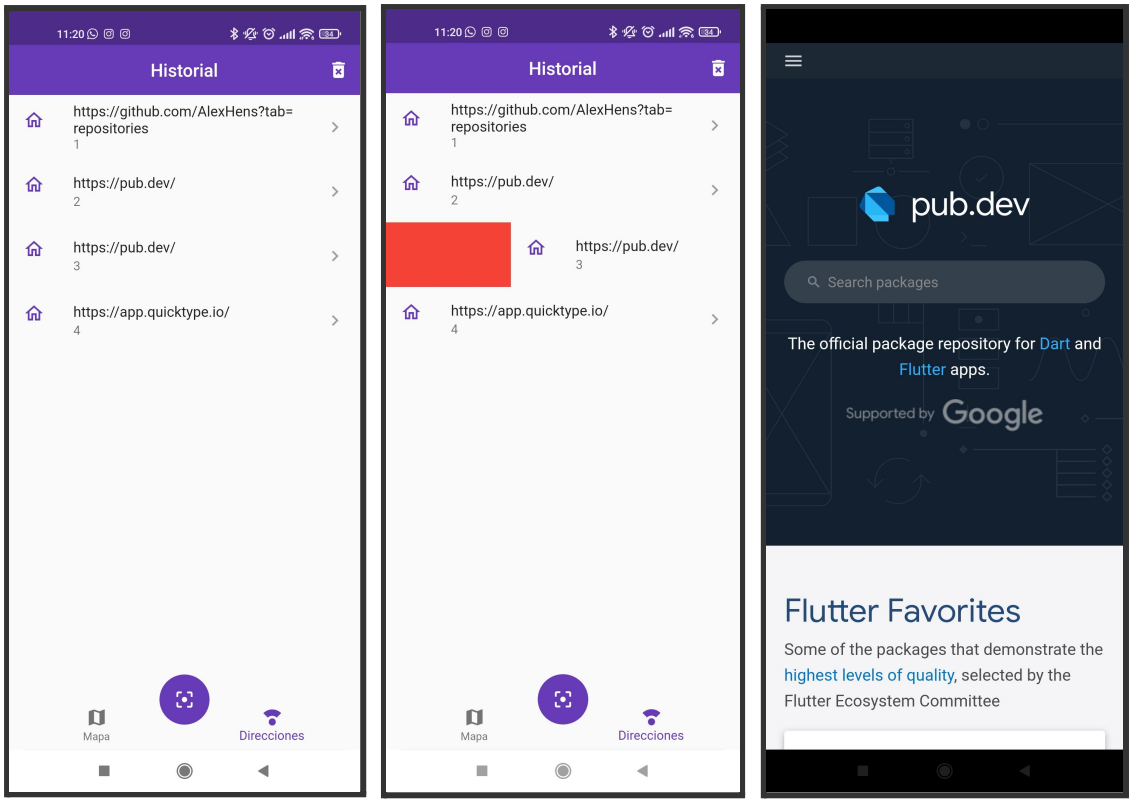


Como podemos ver en la última imágen, al acceder a una ubicación escaneada se nos abre una pantalla con la ubicación en Google Maps. En dicha pantalla tenemos, en la parte superior izquierda un botón para volver a la pantalla anterior, en la parte superior derecha un botón que nos devuelve a nuestra ubicación inicial en el mapa en el caso de que queramos explorar el mismo y en la parte inferior derecha tenemos un botón que nos permite cambiar el tipo de mapa que se está mostrando, los tipos de mapa actualmente disponibles son **mapa normal** y **mapa satélite**.



Por último, analizaremos la sección de los escaneos correspondientes a las direcciones de páginas web de Internet. Dicha sección es accesible desde el botón situado en la parte inferior derecha de la pantalla, la cual nos muestra un icono de señal de wifi y un texto debajo del mismo que dice “Direcciones”.

Una vez hemos accedido a esta pantalla, podemos observar un historial con los escaneos realizados hasta la fecha. Las funcionalidades que nos presentan son las mismas que las del apartado anterior, es decir, podemos acceder a un enlace anteriormente escaneado o eliminar un escaneo determinado.



3. Herramientas usadas para el desarrollo de la App

Para el desarrollo de la aplicación hemos usado las siguientes herramientas:

- **Visual Studio Code**

  Hemos utilizado esta herramienta como nuestro principal editor de textos. Para poder realizar el correcto desarrollo de la aplicación hemos instalado en dicho editor las siguientes extensiones:

  - **Awesome Flutter Snippets:** Esta extensión nos provee de una serie de *snippets*, es decir, un conjunto de clases y métodos que nos permiten programar de una forma más rápida y sencilla, ya que nos proveerá de las plantillas necesarias para implementar el funcionamiento de ciertos componentes de Flutter.
  - **Better Comments:** Es una extensión que simplemente nos permite definir comentarios con diferentes estilos para poder así resaltar ciertas partes del código.
  - **Dart:** Nos provee de un apoyo para el desarrollo y depuración en este lenguaje.

    - **Flutter:** Nos provee de un apoyo para el desarrollo y depuración de aplicaciones con esta herramienta.
    - **Lorem Ipsum:** Simplemente nos permite insertar bloques de texto para probar la estética o diseño de determinados bloques de información.
    - **Terminal:** Esta extensión nos proporciona un terminal accesible desde Visual Studio Code y gracias al cual podemos ver las diferentes salidas de nuestra aplicación.

- **Android Studio**

  Hemos hecho uso de este IDE no para desarrollar directamente en él nuestra aplicación, cosa que podríamos haber hecho, ya que para ello ya tenemos Visual Studio Code, sino que para que nos provea de un emulador de un dispositivo Android para realizar las diferentes pruebas de funcionamiento de nuestra aplicación.

- **Postman**

  Esta es una aplicación que nos permite realizar pruebas API, es decir, nos ha permitido probar el correcto funcionamiento de las diferentes *peticiones HTTP* que haremos para acceder a cierta información.

- **Quicktype.io**

  Esta página, la cual también tiene una extensión para Visual Studio Code, nos ha proporcionado una gran ayuda para mapear los resultados de las diferentes peticiones HTTP que realizamos durante el desarrollo de la aplicación y así saber cómo tratar la información que nos llega de dichas peticiones.

- **Pub.dev**

  Es una página web que nos provee de paquetes oficiales de Dart que nos ayudarán al desarrollo de ciertas funcionalidades de nuestra aplicación. Los paquetes que hemos usado son:

    - **Provider:** Es un paquete que nos permite gestionar de una forma más cómoda y sencilla las distintas peticiones o cambios que hagamos en las pantallas de nuestra aplicación como resultado de la ejecución de ciertas funcionalidades.
    - **Flutter_barcode_scanner:** Es un plugin que será el que nos permita acceder a la cámara del dispositivo y realizar el escaneo de códigos QR, así como obtener el resultado de dichos escaneos.
    - **Path_provider:** Es un plugin de Flutter que nos permite encontrar ubicaciones concretas en el sistema de archivos.

- **Sqflite:** Es un plugin que nos permite crear y gestionar nuestras bases de datos, así como la información que hay en ellas.
- **Url_launcher:** Es un plugin que nos permite acceder en un navegador a los enlaces escaneados con nuestra aplicación.
- **Google_maps_flutter:** Es un plugin que nos permitirá acceder a las diferentes ubicaciones de Google Maps que se han escaneado con nuestra aplicación.

- **Google Cloud Platform**

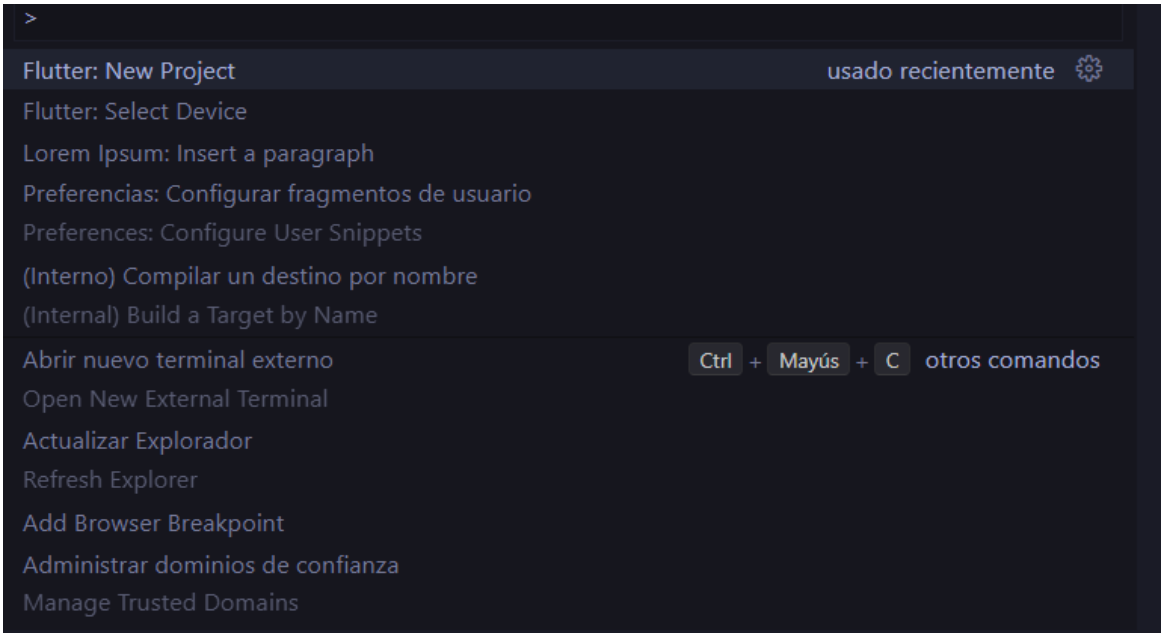
  Es una plataforma que nos provee de las infraestructuras necesarias para crear nuestras claves API, mediante las cuales podremos hacer uso de los diferentes servicios de Google Maps en nuestra aplicación.

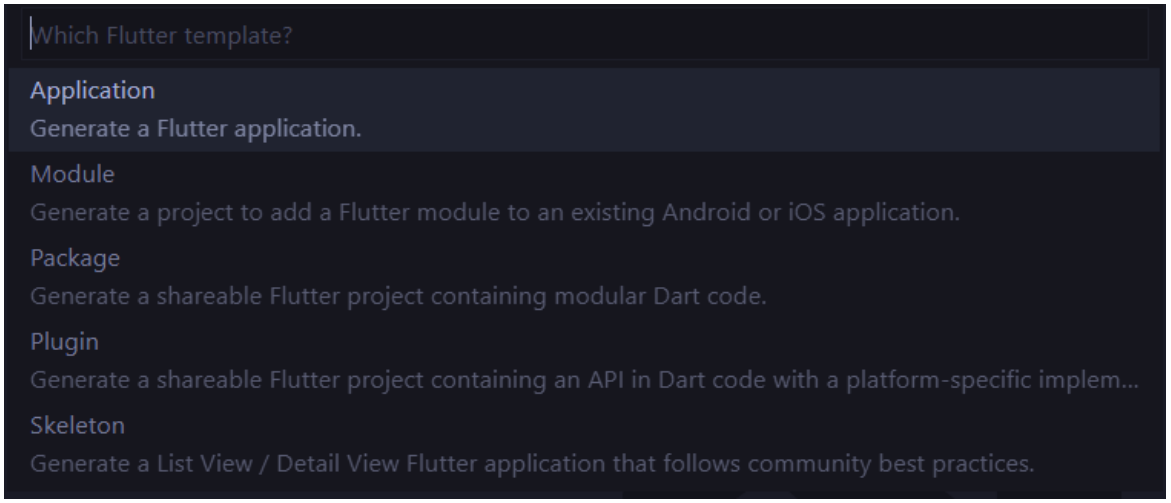
- **DB Browser for SQLite**

  Es una herramienta que nos permite observar de una forma más sencilla si los cambios que estamos realizando en nuestra base de datos se están aplicando correctamente.

4. Proceso de Desarrollo

El primer paso para el desarrollo de nuestra aplicación es crear nuestro proyecto de **flutter**. Para ello usaremos el atajo de teclado `Ctrl + P` y se nos abrirá una nueva pantalla en la que seleccionaremos la opción de `Flutter: New Project` y seguidamente `Application: Generate a Flutter application` .

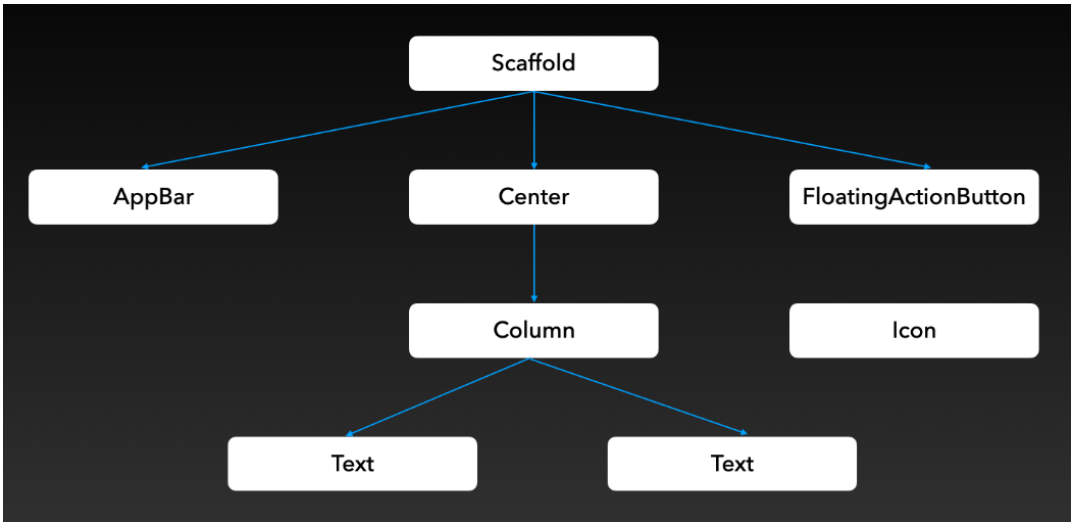




En flutter es más eficiente utilizar **widgets** a métodos privados, porque sólo los va a dibujar de manera condicional cuando es necesario. Un **widget** es una clase que puede tener argumentos posicionales y argumentos con nombre. Son como "piezas" que ya tienen una funcionalidad en específico y se unen para ir creando nuestra aplicación.
Hay dos tipos de widgets que vamos a utilizar durante todo el proyecto:

- "*Staless Widget*" → O también llamados sin estado, ya que estos no van a poder cambiar.
- "*Stateful Widget*" → Con estado, pueden redibujarse así mismo.

Por último mencionar los principales widget que vamos a utilizar:



- **Scaffold:** Corresponde a toda la pantalla en general que podemos visualizar.
- **AppBar:** Muestra una barra en la parte superior de la aplicación
- **Center:** Es un widget que te permite centrar el contenido.

- **Column:** Te permite disponer de una lista vertical de widget.
- **FloatingActionButton:** Te permite dibujar un botón en la pantalla.

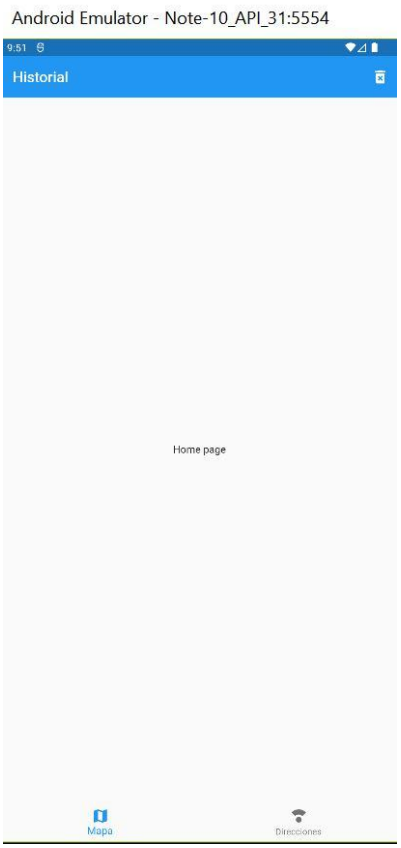
Una vez hemos hecho esta aclaración de flutter y haber creado nuestro proyecto ya podemos empezar a escribir las líneas de código.

Vamos a utilizar únicamente una pantalla, el "*HomePage*" con una estructura como la siguiente:

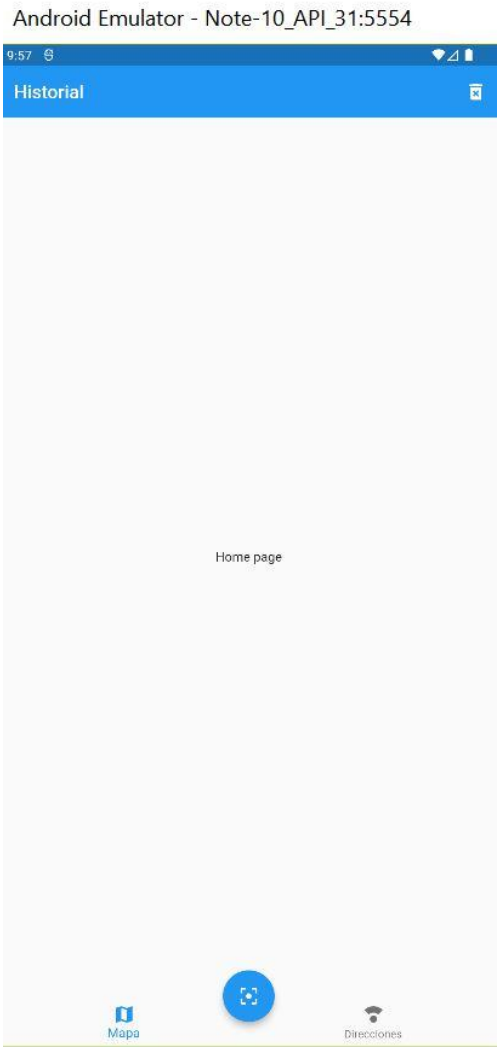
```
class HomePage extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return  Scaffold(
      appBar: AppBar(
        elevation: 0,
        title: const Text('Historial'),
        actions: [
          IconButton(
            icon: const Icon(Icons.delete_forever),
            onPressed: (){
```

Todas las pantallas van a disponer de la misma estructura.
El primer widget que vamos a crear para tenerlo todo de una manera más organizada es el "*CustomNavigationBar*" que va a ser nuestro botón centrar con el que vamos a poder realizar el escanéo. Dentro vamos a tener un "BottomNavigationBarItem" que necesita la creación de dos ítem los cuales serán el primero para mostrar las coordenadas y el segundo para mostrar direcciones URL.

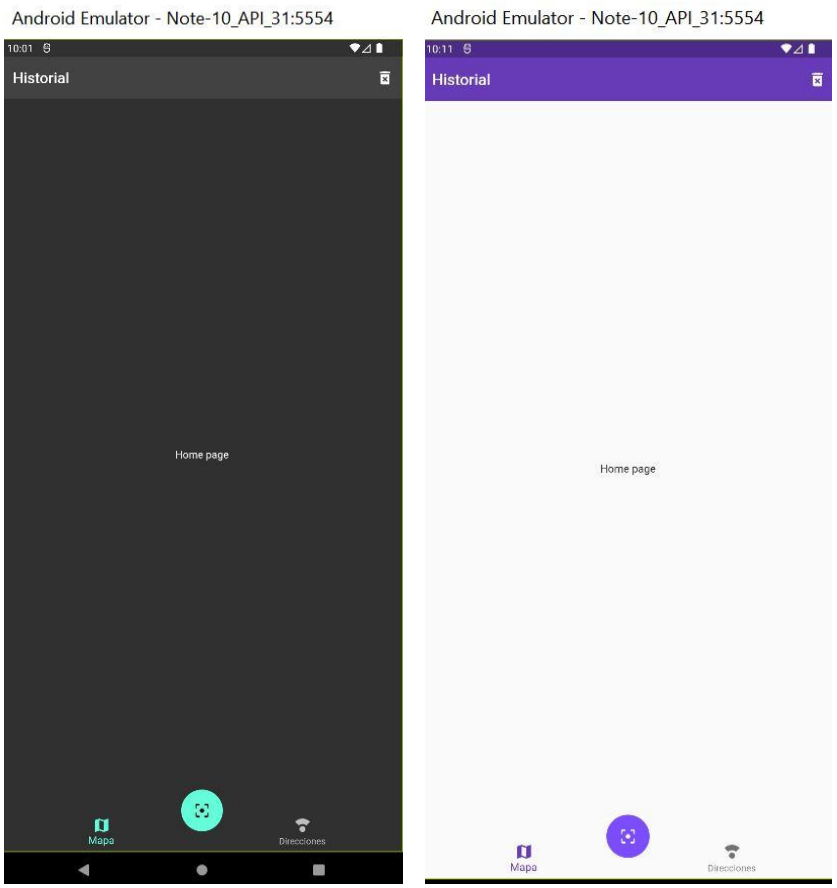


A continuación creamos el "*floatingActionButton*" para poder realizar los escaneos. También será un widget que en nuestro caso se llamará "**scan_button.dart".**
Con la ayuda de "*FloatingActionButtonLocation*" ayudamos a centrar nuestro botón (.centerDocked).



Tema de la aplicación

Vamos a crear nuestro propio tema. En el archivo main hay temas por defecto y podemos utilizar tanto claro como el oscuro, pero nosotros vamos a utilizar el nuestro propio.



Cambiar de página dependiendo del tab

Vamos a crear dos nuevas páginas: "*mapas_page.dart*" y "*direciones_page.dart*".
¿Cómo podemos hacer para cambiar nuestra pantalla de manera condicional a alguna de estás páginas?
Creamos en el home un widget privado sólo va a ser visible en este archivo, "*_HomePageBody*"va a ser el que vamos a colocar en el body de nuestra clase.
Creamos una estructura condicional switch.

```
class _HomePageBody extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
      switch( currentIndex ){
        case 0:
          return MapasPage();
          break;

      case 1:
        return DireccionesPage();
        break;

      default:
        return MapasPage();
    }
  }
```

Para poder cambiar de 0 a 1 vamos a utilizar un **"gestor de estados"** en nuestro caso "*Provider*", vamos a poder redibujar el widget cuando nosotros queramos.
Es necesaria la instalación de `provider: ^6.0.2` en nuestro archivo "*pubspec.yaml*" el cual debemos tratar con especial cuidado porque hasta las tabulaciones son importantes en este archivo.

Ahora ya creamos un nuevo archivo llamado "*ui_provider.dart*" que va a ser una simple clase con una única propiedad int _selectedMenuOpt = 0; es una propiedad privada y la única manera de cambiarlo va a ser mediante getters y setters.

En esta clase vamos a utilizar: notifyListeners() que informa a todos los widgets que ha cambiado su valor.

El siguiente paso es envolver nuestro *MaterialApp* del main en un nuevo widget: Multiprovider. En este momento en nuestro árbol de widget(context) vamos a tener una instancia de UiProvider.Ahora necesitamos poder leer la instancia para cambiar todo de manera condicional.

Por último hay que cambiar el valor del ui_provider de manera dinámica.

Cambiar la opción del menú

Accedemos a "*custom_navigatorbar*" y creamos:
final currentIndex = uiProvider.selectedMenuOpt;

Leer un código QR

Necesitamos instalar `flutter_barcode_scanner: ^2.0.0` en nuestro archivo "*pubspec.yaml*".

```
import 'package:flutter_barcode_scanner/flutter_barcode_scanner.dart';
```

```
String barcodeScanRes = await
FlutterBarcodeScanner.scanBarcode('#3D8BEF','Cancelar', false,
ScanMode.QR);
```

SQFLite

Lo primero que necesitamos es el `path_provider: ^2.0.9` nos sirve para saber dónde se encuentra el archivo en la base de datos, vamos a saber el path exacto de donde se

encuentra. La siguiente instalación es la de `sqflite: ^2.0.2+1`. Por último, debemos instalarnos la herramienta DB Browser for SQLite, mediante la cual podremos asegurarnos del correcto funcionamiento de las operaciones realizadas con la base de datos.
Después de terminar todas las instalaciones anteriores, lo primero que debemos hacer es crear nuestro nuevo archivo "*db_provider.dart*" en la carpeta de "*providers*". En este archivo crearemos nuestra base de datos e implementaremos las distintas operaciones que vamos a realizar con la base de datos en nuestra aplicación.

- **Creación de la base de datos:** Para la creación de la base de datos crear una función que nos ayude a crear e iniciar la base de datos. Dicha función es la siguiente:

```dart
Future<Database> initDB() async{

  // Path donde almacenamos la base de datos
  Directory documentsDirectory = await getApplicationDocumentsDirectory();
  final path = join(documentsDirectory.path, 'ScansDB.db');
  print(path);

  // Crear base de datos
  return await openDatabase(
    path,
    version: 1,
    onOpen: (db) {},
    onCreate: (Database db, int version) async{

      await db.execute('''

        CREATE TABLE Scans(
          id INTEGER PRIMARY KEY,
          tipo TEXT,
          valor TEXT
        );

      ''');

    }
  );

}
```

  El código anterior comienza con el provider de la base de datos obteniendo la ruta del directorio en el que se alojará nuestra base de datos. Después, crearemos el archivo "*ScansDB.db*", el cual corresponde al archivo de la base de datos. Por último, retornamos la base de datos, especificando los datos como la versión, el path en el que se aloja, el código que se deberá ejecutar al abrir la base de datos, el cual corresponde al provider de la misma, y el código que se ejecutará al crearse esta, el cual es la creación de la tabla "*Scans*", en la cual guardaremos los diferentes atributos de nuestros escaneos(identificador, tipo y valor).

- **Añadir un nuevo registro de escaneo:** Para añadir un nuevo registro debemos crear dos variables, la primera la vamos a llamar db la cual será la encargada de realizar un *await* al database, y una segunda variable denominada *res* que hará el

segundo await y realiza la inserción de manera automática únicamente pasándole el nombre de la tabla 'Scans" y el ScanModel del argumento.

```dart
Future<int> nuevoScan( ScanModel nuevoScan) async {

  final db = await database;
  final res = await db.insert('Scans', nuevoScan.toJson() );
  print(res);

  // Es el ID del último registro insertado
  return res;
}
```

- **Obtener el registro de escaneos:** Vamos a realizar tres formas diferentes de obtener los registros de los escaneos. Una primera, a través de su Id, la segunda se obtienen los registros de todos los scans y la tercera se obtienen por tipo.

```dart
Future<ScanModel?> getScanById ( int id) async{

  final db = await database;
  final res = await db.query('Scans', where:'id = ?', whereArgs: [id]);

  return res.isNotEmpty ? ScanModel.fromJson( res.first ):null;
}

Future<List<ScanModel>> getTodosLosScans () async{

  final db = await database;
  final res = await db.query('Scans');

  return res.isNotEmpty
  ? res.map((s) => ScanModel.fromJson(s)).toList()
  :[];
}

Future<List<dynamic>> getScansPorTipo ( String tipo) async{

  final db = await database;
  final res = await db.rawQuery('''
    SELECT * FROM Scans WHERE tipo = '$tipo'
  ''');

  return res.isNotEmpty
  ? res.map((s) => ScanModel.fromJson(s)).toList()
  :[];
}
```

Tenemos que realizar lo mismo que en el paso anterior pero esta vez será un "*query*" en lugar de una inserción y le añadimos una condición con la ayuda del *where*.

Para llamar al return hemos añadido una nueva condición para comprobar si este está vacío o no.
En el "*getScansPorTipo*" hemos querido realizarlo de la manera tradicional realizando la consulta directamente a mano para comprobar su correcto funcionamiento.

- **Actualizar un registro de escáner determinado:** Para actualizar el registro de un determinado escaneo, implementaremos la siguiente función:

```
Future<int> updateScan(ScanModel nuevoScan) async{
  final db = await database;
  final res = await db!.update('Scans', nuevoScan.toJson(), where: 'id = ?', whereArgs: [nuevoScan.id]);

  return res;
}
```

Como vemos en la función anterior, nuestra función recibe como parámetro un registro de escaneo, el cual sustituirá al ya existente en la base de datos. Lo reemplazamos gracias a los distintos métodos que nos proporciona el provider de la base de datos. Como podemos ver en la segunda sentencia de nuestra función, haremos el update de ese registro especificando la tabla a la que pertenece y la condición que tiene que cumplir, en nuestro caso que el identificador sea el mismo.

- **Borrar registros de escáneres:** Para borrar los registros tenemos dos opciones, borrar un único scan o borrarlos todos directamente.
La primera opción se realiza de la misma forma que todos los anteriores pero añadiendo la propiedad "*detele*". La segunda opción también la hemos querido realizar de la manera tradicional para comprobar su correcto funcionamiento.

```dart
Future<int> deleteScan( int id) async {

  final db = await database;
  final res = await db.delete('Scans', where: 'id = ?', whereArgs: [id]);

  return res;
}

Future<int> deleteAllScans() async {

  final db = await database;
  final res = await db.rawDelete('''
    DELETE FROM Scans
  ''');
  return res;
}
```

Escanear un código QR

Para escanear un código QR haremos uso de nuestro widget "*ScanButton()*", el cual funciona de la siguiente manera:

```dart
class ScanButton extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return FloatingActionButton(
      child: Icon(Icons.filter_center_focus),
      elevation: 0,
      onPressed: () async{

        String barcodeScanRes = await FlutterBarcodeScanner.scanBarcode('#3D8BEF', 'Cancelar', false, ScanMode.QR);
        //final barcodeScanRes = 'https://github.com/AlexHens?tab=repositories';
        //final barcodeScanRes = 'geo:45.287135,-75.920226';

        if(barcodeScanRes == -1){
          return ;
        }

        final scanListProvider = Provider.of<ScanListProvider>(context, listen: false);
        final nuevoScan = await scanListProvider.nuevoScan(barcodeScanRes);
        //scanListProvider.nuevoScan('geo:15.33,15.66');
        launchUrlPropio(context, nuevoScan);


      }
    ); // FloatingActionButton
  }
}
```

Como podemos observar en el código anterior, cuando el botón sea pulsado se ejecutará todo el código de la sección "*onPressed()*".

El código comienza pidiendo que se realice un escaneo gracias a los recursos que nos provee el paquete **Flutter_barcode_scanner**, esa función nos retornará un determinado código. Si el código retornado es igual a -1, entonces quiere decir que se ha

cancelado el escaneo. En otro caso, declaramos un provider que escuche a la espera de la llegada de información y la almacenamos en un nuevo registro para luego lanzar el enlace con la función siguiente:

```
launchUrlPropio(BuildContext context, ScanModel scan) async {

  final url = Uri.parse(scan.valor);

  if(scan.tipo == 'http'){
    if (!await launchUrl(url)) throw 'Could not launch $url';
  }
  else{
    Navigator.pushNamed(context, 'mapa', arguments: scan);
  }
}
```

Como podemos ver, lo primero que hacemos es traducir a una url válida el valor del escaneo que hemos recibido y después miramos qué tipo de escaneo hemos realizado.

En el caso de que leamos un escáner de tipo **http** debemos llamar a la función **launchUrl**, la cual ha sido obtenida del paquete **url_launcher**, especificando la url que hemos recibido. Una vez llamada esta función se procederá a abrir el enlace especificado en el navegador.

En el caso de que el escáner leído sea de tipo **geo** saltaremos a la pantalla que nos mostrará la ubicación en Google Maps de las coordenadas escaneadas. En dicha pantalla mostraremos la ubicación en Google Maps gracias a los servicios de los que nos provee el paquete **google_maps_flutter**. Para ello, debemos crear nuestras API keys para poder acceder a la ubicación de Google Maps desde nuestra propia aplicación. Para crear dichas keys, debemos acceder a la página de **Google Cloud Platform > Credenciales**, debemos crear una para IOS y otra para Android, y quedaría finalmente así:

Una vez, hemos creado las keys, en nuestro caso solo hemos usado la de Android, por tanto hemos realizado las configuraciones necesarias para Android. Dichas configuraciones consisten en añadir las siguientes líneas de código a nuestro archivo **AndroidManifest.xml**, situado en la carpeta **android > app > src > main**, junto con la key correspondiente que hemos creado anteriormente.

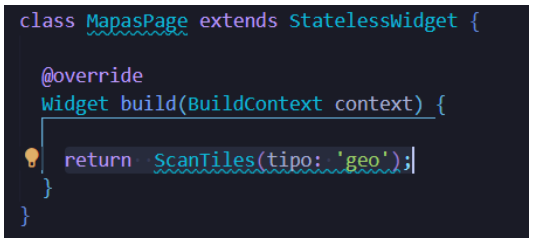
Por último, una vez terminada esta configuración, procederemos a implementar su funcionamiento en nuestra aplicación. Para dicha implementación necesitaremos especificar el punto inicial en el que se va a colocar la cámara, el tipo de mapa que mostraremos y el código que se ejecutará durante la creación del mapa, el cual se creará automáticamente gracias a los controladores que nos proveen el paquete de **google_maps_flutter**.

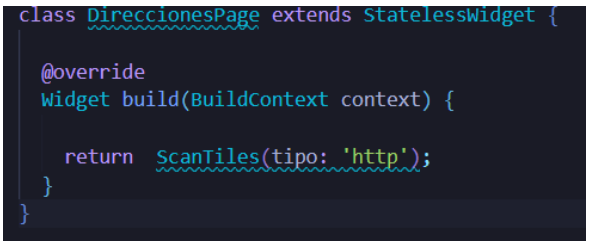
Historial

Para poder crear el historial de nuestra aplicación tanto de los mapas como de las direcciones hemos creado una nueva clase denominada "*ScanTiles*".
La función devolverá un *ListView.builder* que será la encargada de decidir qué información irá a una página u a otra. Para ello debemos crear una clave que sea única.
Dentro de esta misma función hemos construido en el ítem builder una clase "*Dismissible*", que contendrá un child encargado de poner los valores que queremos mostrar en la pantalla, los distintos iconos y realizar la llamada a la función "*launchURL*" para acceder a la información que le estamos proporcionando.

```
return ListView.builder(
  itemCount: scans.length,
  itemBuilder: (_ , i) => Dismissible(
    key: UniqueKey(),
    background: Container(
      color: Colors.red,
    ), // Container
    onDismissed: (DismissDirection direction){
      Provider.of<ScanListProvider>(context, listen: false)
      .borrarScanPorId(scans[i].id!);
    },
    child: ListTile(
      leading: Icon(
        this.tipo == 'http'
          ? Icons.home_outlined
          : Icons.map_outlined,
        color:Theme.of(context).primaryColor
        ), // Icon
      title: Text(scans[i].valor),
      subtitle: Text (scans[i].id.toString()),
      trailing: Icon(Icons.keyboard_arrow_right, color:Colors.grey),
      onTap: () => launchURL(context, scans[i])
    ), // ListTile
  ) // Dismissible
); // ListView.builder
```

Ya sólo nos faltaría llamar a la función en ambas páginas, tanto como en mapas y direcciones.





Conclusiones

El objetivo del desarrollo de esta aplicación es el de ofrecer una cierta facilidad a la hora de compartir y acceder información. Mientras pensábamos en por qué deberíamos desarrollar esta aplicación nos dimos cuenta de la cantidad de aplicaciones que podría tener, es decir, podría usarse tanto para pasar lista en colegios, institutos o universidades, como método de identificación en muchos trabajos, para compartir apuntes o documentos en general, o para acceder a ciertos datos de otras personas, por ejemplo en el ámbito médico a la hora de acceder al historial médico de ciertos pacientes.

Por otro lado, el desarrollo de esta aplicación nos ha servido para aprender a usar ciertas herramientas y comprobar de primera mano el esfuerzo y trabajo que supone desarrollar una aplicación real y 100% funcional como lo es esta.

Por último, tenemos que decir que para nosotros el desarrollo de este proyecto ha supuesto un gran reto en el que hemos aprendido muchísimo y en el que al final hemos visto un resultado de todos nuestros esfuerzos y una aplicación de todos nuestros conocimientos.