Flutter y FireBase CRUD

https://www.youtube.com/playlist?list=PLutrh4gv1YI8ap4JO23lN81JOSZJ3i5OO

Vídeo 30. CRUD (conectando con firebase)

- Creamos nuestro flutter con **flutter create --org com.agrgal crud**
- Ir a firebase.google.com, consola y con nuestro gmail
    - Creamos nuevo proyecto, que le pongo crud-test
    - No vamos a necesitar Google Analytics, pero lo recomienda para algo real.
    - Usamos la instancia Firestore

- Deberíamos ponerla en modo de producción cuando la aplicación sea real. Por ahora en modo de pruebas y podemos ampliar la fecha.
    - En reglas, amplío la fecha un año, para no tener problemas. Publicamos.
- Escoger un servidor cerca de donde estemos. Yo escojo eur3, que es multirregional.

- Tenemos colecciones. Por ejemplo, "**people**". Se pueden crear documentos, como registros, de ID automático. Crearé dos, con un campo **name** los dos, y valores Aurelio y Juan.

Vamos a usar básicamente la herramienta FlutterFire para conectar la base de datos de FireStore a nuestro proyecto.

1. En nuestro proyecto, instalar desde la línea de comandos **flutter pub add firebase_core.** Actualizo lo que puedo.
2. ***Tenemos que instalar Firebase CLI***. Seguimos las instrucciones de https://firebase.google.com/docs/cli?hl=es#sign-in-test-cli para tu sistema operativo.
    a. Instalar **curl -sL https://firebase.tools | bash** (pero si lo tengo instalado de antes, parece que me pide actualizarlo. Necesito sudo). Firebase tools se queda instalado en el equipo. *En el vídeo prefieren el método de instalación npm.*
        i. **firebase --version** para comprobar la versión que está instalada.
    b. Hacemos **firebase login** (puede que me diga que ya estoy logueado). Si no, se abre la ventana del navegador y debo conceder el acceso.
    c. Y si hago **firebase projects:list** , me muestra una lista de los proyectos que tengo abiertos.
3. Instalamos **flutterfire CLI** con **dart pub global activate flutterfire_cli** .
4. Y configuramos con **flutterfire configure**

Seguimos con la configuración según las preguntas que me hace. Usar flechas y espacios para la selección. Enter para escoger.

- Escojo mi proyecto. En este caso crud-test.
- En el siguiente paso me indica para qué plataformas usaré el proyecto. Las cogemos todas (eso dice el vídeo).
- Puede preguntarme por actualizaciones. A mí me pregunta el id de la aplicación android o el nombre del paquete. En el vídeo no. En fin, le pongo **com.agrgal.crud**
- Cuando termina, en la carpeta lib agrega un fichero llamado **firebase_options.dart**

- NOTA: da un error en la consola de depuración. Nos dice que la versión de kotlin es incorrecta. Mirar qué versión necesita. Y dentro de la carpeta **android**, en el fichero **settings.gradle** actualizar la versión de **kotlin** → id "org.jetbrains.kotlin.android" version "1.8.22" apply false (tenía la 1.7.10)

- Ahora tenemos que importar **firebase_core** y el fichero **firebase_options.dart** en nuestro **main.dart**.

```
// importaciones
import 'package:firebase_core/firebase_core.dart';
import 'firebase_options.dart';
```

- Y cambiamos ***main*** para que corra de la siguiente manera:

```
void main() async { → función que trabaja asíncrona.
  WidgetsFlutterBinding.ensureInitialized(); → me aseguro de inicializar los widgets
  await Firebase.initializeApp( → conecto con Firebase según sus opciones
        options: DefaultFirebaseOptions.currentPlatform,
  );
  runApp(MyApp()); → inicio mi aplicación.
}
```

Si logra correr la aplicación hemos completado el primer paso: ***conectar con firebase***

Vídeo 31. CRUD (Leer datos)

- Hemos conectado al servicio ***FireBase***, pero no a su base de datos ***FireStore***.
- Para conectarme a **FireStore** debemos instalar el paquete cloud_firestore y seguimos las instrucciones de https://pub.dev/packages/cloud_firestore/install → **flutter pub add cloud_firestore**

NOTA: no me da el problema de la versión mínima de Sdk. Si no, hay que cambiarlo en android>>app>>build.gradle , en defaultConfig, y poner la versión que me recomienda el error.

NOTA: si me da el error de 64k, en el mismo fichero android>>app>>build.gradle, en defaultConfig se añade una línea multiDexEnabled true (se lo he puesto por si acaso y sigue funcionando).

- Creamos una carpeta nueva que se llame **services** y dentro de ella ponemos un fichero **firebase_service.dart**, que administrará nuestra base de datos.
- En este fichero hacemos la importación → **import "package:cloud_firestore/cloud_firestore.dart";**

```
import "package:cloud_firestore/cloud_firestore.dart";

FirebaseFirestore db =FirebaseFirestore.instance; → instancia de la base de datos

Future<List> getPeople() async {  →  Función asíncrona que devuelve los datos
  List people = []; →  inicializo  una lista vacía
  CollectionReference crpeople = db.collection("people");
                  →  creo una colección crpeople desde la base de datos.
 QuerySnapshot qspeople = await crpeople.get();  → importante, await.
                  → obtiene todos los documentos de la colección. Si hay muchos conviene
                  hacer un filtrado.

  for (var documento in qspeople.docs) {  → por cada documento
        people.add(documento.data()); → añadimos los datos a la lista
  }

  return people;  → retorna la lista.
}
```

- Y ya trabajamos en **main** para recuperar estos datos.
- El Scaffold lo extraemos (lo podemos llamar **HOME**) y lo convertimos en **Stateful**. Porque tiene que actualizarse.

```
return Scaffold(
        appBar: AppBar(
                title: const Text('Lectura de la base de datos'),
        ),
        body: FutureBuilder (
                future: getPeople(), → lo que retorna la función en el futuro…
                builder: (context, snapshot) { → cuando retorne, lo que construye
                        if (snapshot.hasData) { → si tiene datos.
                                return ListView.builder( → construye una lista
                                itemCount: snapshot.data?.length, → nº de items data? → si no es nulo
                                itemBuilder: (context, index) { → construye cada item
                                        return Text(snapshot.data?[index]['name'] ?? "");
                                        → retorna el dato con el nombre name
                                        data? → si no es nulo
                                        ?? "" → si no hay nada, que imprima comillas vacías.
                                });
                        } else { → si NO tiene datos. O no los ha cargado aún…
                                return const Center( → Dibujo del progreso de carga.
                                        child: CircularProgressIndicator(
                                        color: Colors.red,
                                        ),
                                        );
                        }
                },
        ),
        );
```

- Si en la función getPeople() añado, justo antes de retornar la lista, estas líneas….

```
await Future.delayed(
        const Duration(seconds: 2),
);
```

- Se retrasa dos segundos la entrega de la lista, y en el Futurebuilder del Scaffold se puede ver lo que hay cuando no hay datos: el círculo de carga.

Vídeo 32. CRUD (Guardar datos)

- Lo primero es guardar el home en una página aparte. Reestructurar los imports.

  En el **main.dart** se añaden rutas

```
Widget build(BuildContext context) {
        return MaterialApp(
        title: 'Material App',
        initialRoute: "/",
        routes: {
                "/": (context) => const Home(),
                "/add": (context) => const AddNamePage(),
        },
        );
```

- En la página **home_page.dart** se almacena el Home. Reestructurar imports. Se añade un floatingActionButton para pasar a la página de añadir.

```
floatingActionButton: FloatingActionButton(
        onPressed: (){
          Navigator.pushNamed(context, "/add"); → navegar hasta la página de añadir.
        },
        child: const Icon(Icons.add)),
```

- Se crea una página **add_name_page.dart** donde se inserta este Scaffold, dentro de un SatefulWidget. El objetivo es poder añadir un nombre.

```
return Scaffold(
    appBar: AppBar(
        title: const Text("Añadir Nombre a la BD")
    ),
    body: Padding(
      padding: const EdgeInsets.all(15.0),
      child: Column (
        children: [
          const TextField(
            decoration: InputDecoration(
              hintText: "Introduce el nombre",
            ),
          ),
ElevatedButton(
            onPressed: (){
            },
            child: const Text("Guardar"))
        ],
      ),
    )
  );
```

- en el **add_name_page.dart** añadimos una variable **TextEditingController nameController = TextEditingController(text: "");** antes del Widget build. Y en el TextField añadimos la propiedad **controller: nameController**. Esa será la variable que admite nuestro campo.

Por fin, debemos hacer un servicio que guarde la información en la base de datos de Firebase FireStore.

```
// Grabar datos
Future<void> addPeople(String name) async{
        → función que recibe un String y no devuelve nada. Es asíncrona.
 await db.collection("people").add( → esperar y añadir a la colección un documento
   {"name": name}, → En formato parecido a JSON
 );
}
```

Y… la llamamos desde el ElevatedButton de la página **add_name_page.dart**

```
ElevatedButton(
        onPressed: () async { → debe ser asíncrona
                // print(nameController.text);
                addPeople(nameController.text).then((_) { → función entonces, cuando termina…
                  Navigator.pop(context); → es cuando llama a la página anterior
                });
                child: const Text("Guardar"))
```

Y para que recargue la página y los datos es suficiente con, en la llamada del floatingActionButton…

```
floatingActionButton: FloatingActionButton(
  onPressed: () async { → convertirla en asíncrona
    await Navigator.pushNamed(context, "/add"); → esperar a que lo añada
    setState(() {}); → recargar el estado.
  },
  child: const Icon(Icons.add)),
```

Vídeo 33. Actualizar en FireStore

- Copiamos la página **add_name_page.dart** a **edit_name_page.dart.** Vamos a reutilizarla, pero haciendo los cambios para que sirva de edición.

  NOTA: Dentro de un widget Stful o Stless, pulse sobre el nombre y F2. El nombre se cambiará en todo el documento.

- Añado la ruta: **"/edit": (context) => const EditNamePage(),** en el **main.dart.** Comprobar que se ha añadido el import de la página.

- En el home page, en vez de que el ListView.builder retorne una lista de objetos de Texto, hago que retorne un ListTile:

```
return ListTile(
  title:Text(snapshot.data?[index]['name']),
  onTap: (){ → Tengo una nueva funcionalidad
    Navigator.pushNamed(context, "/edit",
    arguments: { → Y además, puedo pasar argumentos.
      "nombre":snapshot.data?[index]['name'],
    });
  },
```

- En la nueva página de edición, **edit_name_page.dart**
- Declaro la variable Map (diccionario) **final Map misArgumentos = ModalRoute.of(context)!.settings.arguments as Map;** bajo el Widget Builder. Recibirá los datos que se le pase a través del Navigator.
- Y la propiedad **hintText: misArgumentos["nombre"]**, para que se muestre el texto que hemos marcado. Pero… él no lo hace así. Lo que hace es:
  - Tengo una variable tipo controller llamada **nameController**
  - A continuación de la definición de misArgumentos, escribe **nameController.text = misArgumentos["nombre"];**
  - Y en el textField, como **controller**: **nameController,** ya aparecería el nombre.

En nuestro fichero firebase_service.dart

1º) Agregamos una función para hacer un cambio…

```
// Actualizar datos
Future<void> updatePeople (String uid, String newname) async { → dos parámetros, uid y el nuevo nombre
  await db.collection("people").doc(uid).set({"name":newname});
        → Importante: podemos hacer un set, pero siempre que sepamos en qué id.
}
```

2º) En la función get, hacemos una modificación en cómo recibimos los datos para obtener el id

```
Future<List> getPeople() async { // Función asíncrona que devuelve los datos
  List people = []; // inicializo la lista vacía
   CollectionReference crpeople = db.collection("people"); // creo una coleccion crpeople de la base de datos.
    QuerySnapshot qspeople = await crpeople.get(); // obtiene todos los documentos de la colección
  // QuerySnapshot qspeople = await db.collection("people").get(); --> mirar que tb nos vale
  for (var documento in qspeople.docs) {
        final Map<String,dynamic> datos = documento.data() as Map<String,dynamic>;
              → declaro una variable que recibe un diccionario o mapa de Firebase
        final person = { → construyo otro mapa con los datos que me interesan, entre ellos el uid
        "name": datos["name"],
        "uid": documento.id, → añado uid
        } ;
        people.add(person);  → añado ahora el nuevo mapa.
  }
  await Future.delayed( // forzar que se muestre el elemento de carga
        const Duration(seconds: 2),
  );
  return people; // retorna la lista
}
```

==========

- Entonces en **home page**, dentro del ListTile y del onTap, puedo navegar hasta la página de actualización mandando el uid…

```
return ListTile(
  title:Text(snapshot.data?[index]['name']),
  onTap: () async { → ¡¡Ojo!! es una función asíncrona
    await Navigator.pushNamed(context, "/edit", → await, esperar a que termine.
    arguments: { → le puedo mandar varios argumentos
      "nombre":snapshot.data?[index]['name'],
      "uid": snapshot.data?[index]['uid'],
    });
    setState(() {}); → Asíncrona, y debe actualizar su estado
  },

  },
);
```

- Y en la página de **edit_page**, al pulsar el botón de actualizar…

```
ElevatedButton(
onPressed: () async { → necesitamos que sea asíncrona
        // print(nameController.text);
        // print(misArgumentos['uid']);
        updatePeople(misArgumentos['uid'], nameController.text).then((_){
                Navigator.pop(context); → y entonces, que vuelva al contexto anterior.
        });
},
```

Vídeo 34. CRUD (Borrar datos con flutter)

- En el vídeo nos dice que se puede hacer de muchas maneras. Lo importante es que el usuario debe validar el borrado de los datos.
- El nos recomienda usar un widget llamado **Dismissible**. Envolvemos el **ListTile** con este widget.

- **Dismissible** necesita una clave única. Se puede usar UniqueKey(), para generarla. Pero en este ejercicio ya tenemos una, que es el uid (dentro de Key) , luego

```
return Dismissible(
    key: Key(snapshot.data?[index]['uid']), // comentado UniqueKey(),
    child: ListTile(.....
```

- Dismissible añade la funcionalidad de arrastrar a la derecha y que desaparezca un elemento de ListTile . Pero no desaparece el snapshot, ni el dato en la base de datos. Si recargo, los elementos vuelven a aparecer.

- Añadimos algunos detalles a Dismissible

```
  return Dismissible(
    background: Container( → al deslizar aparecerá en rojo, y con un icono de borrar
      color: Colors.red[300]
      child: const Icon(Icons.delete_sweep_outlined,)
    ),
    direction: DismissDirection.startToEnd, → sólo se mueve de izquierda a derecha
    key: Key(snapshot.data?[index]['uid']), // UniqueKey(),
    ....
```

- Pero debemos añadirle algo importante: confirmDismiss. Recibe una función con un parámetro que es una dirección y devuelve un booleano. Si es true, se borra.

```
return Dismissible(
  confirmDismiss: (direction) async { → Función asíncrona porque es un Future
    print("Estoy borrando");
    return true; → devuelve booleano. Si es true se borrará, y si es false NO.
  },
  background: Container(....
```

- Pero claro, debemos obtenerlo tras pedir confirmación al usuario, a través de una alerta:

```
confirmDismiss: (direction) async {
    bool resultado = false; → en principio es false

    resultado = await showDialog( → en esta variable se almacena true o false
      context: context,
      builder: (_){
        return AlertDialog( → Diálogo de alerta
          title: const Text("¿Estás seguro?"),
          content: Text("¿Estas seguro de que quieres eliminar ${snapshot.data?[index]['name']}"),
          actions: [
            TextButton( → botón que si se presiona devuelve false
              onPressed: (){
                return Navigator.pop(context, false);
              },
              child: const Text("No"),
              ),
              const SizedBox(width: 10.0,),
              TextButton(
              onPressed: (){ → botón que si se presiona devuelve true
                return Navigator.pop(context, true);
              },
              child: const Text("Sí, borrar"),
              ),
          ],); });
    return resultado ; → confirm dismiss retorna el resultado.
  },
```

- Siguiente paso. Función de borrado en **firebase_services.dart**

```
// Eliminar datos
Future<void> deletePeople(String uid) async { → sólo necesita el uid.
  await db.collection("people").doc(uid).delete();
}
```

- Y ahora, en el **Dismissible**, añadir el parámetro **onDismissed**

  return Dismissible(
  onDismissed: (direction) async { → **se ejecuta si se ha confirmado el Dismiss**
  await deletePeople(snapshot.data?[index]['uid']); → **espera hasta efectuar el borrado**
  },
  confirmDismiss: (direction) async {.....

- Pero… Puede dar un error. Intentamos añadir uno nuevo, y simplemente retrocedemos. Salta un error: "FlutterError (A dismissed Dismissible widget is still part of the tree. Make sure to implement the onDismissed handler and to immediately remove the Dismissible widget from the application once that handler has fired.)".

  Porque también hay que borrar el valor de la lista de los datos

  return Dismissible(
  onDismissed: (direction) async {
  await deletePeople(snapshot.data?[index]['uid']);
  snapshot.data?.removeAt(index); → IMPORTANTE
  },
  confirmDismiss: (direction)....