

Programación Multimedia y Dispositivos Móviles

UD 10. Firebase

Javier Carrasco

Curso 2021 / 2022



Este obra está bajo una licencia de Creative Commons Reconocimiento 4.0 Internacional.
Última actualización: septiembre de 2021.

Versión impresa en Amazon: <https://www.amazon.es/dp/B08NM4XV4Q>

Firebase

10. Firebase.....	3
10.1. Primer proyecto <i>Firebase</i>	3
10.2. Crear una base de datos en <i>Firebase</i>	8
Realtime Database.....	9
Cloud Firestore.....	14
10.3. Tipos compuestos en <i>Firebase</i>	24

10. Firebase

Firebase¹ es una API de Google multiplataforma (Android, iOS, Web o Unity) que permite a las aplicaciones trabajar con datos en la nube, guardando y sincronizando en tiempo real.

Firebase dispone de varios tipos de herramientas, analíticas, de desarrollo, monetización, etc, que permitirán hacer crecer a las aplicaciones que las utilicen. Dispone de un plan gratuito que permite trabajar con todo su potencial, pero si, por suerte o por desgracia, la aplicación tiene mucho éxito, se deberá pasar a un plan de pago o plantearse un cambio de API.

Centrando la atención en las herramientas de desarrollo, Firebase permitirá delegar ciertas operaciones, lo cual ahorrará tiempo y código, además de un salto cualitativo en las aplicaciones. Destacar las herramientas de almacenamiento, testeo, configuración remota, mensajería en la nube o autenticación, entre otras.

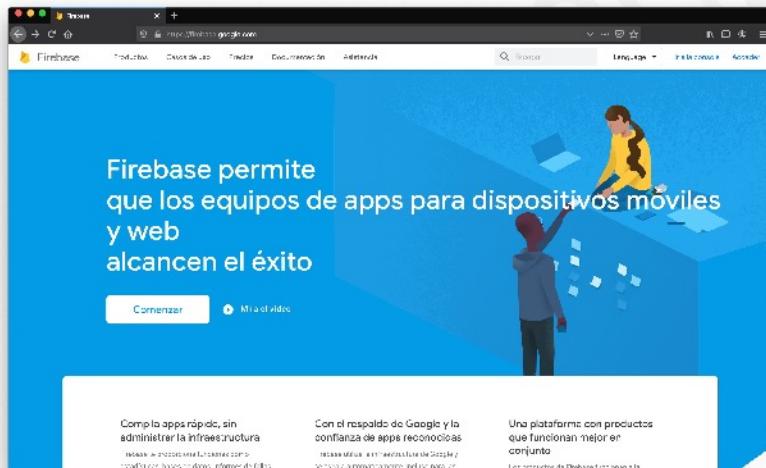


Figura 1

Para poder hacer uso de *Firebase* en las aplicaciones que se creen, se necesitará disponer de una cuenta de Google.

10.1. Primer proyecto *Firebase*

Para iniciar un nuevo proyecto se hará uso de la opción "*Ir a la consola*", o "*Comenzar*", que se encuentra en la web de *Firebase*. Será necesario autenticarse con una cuenta de Google para poder acceder.

Se comenzará por "*Crear proyecto*", proceso muy sencillo que se desarrolla a través de un asistente en la misma web de *Firebase*.

1 Firebase (<https://firebase.google.com/>)

4 UNIDAD 10 FIREBASE

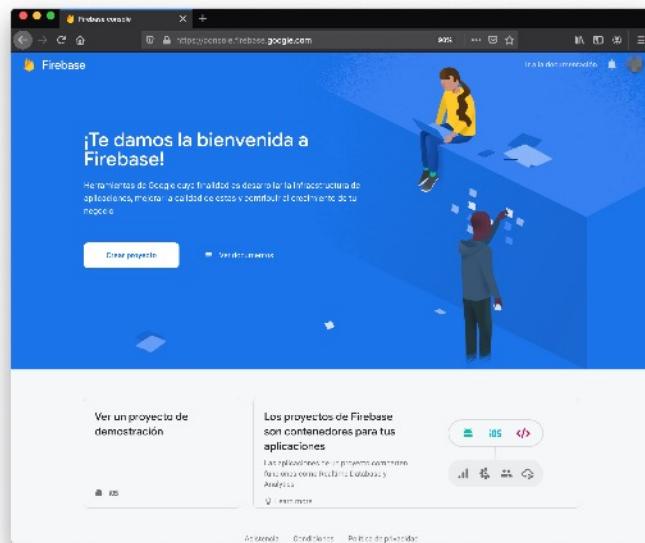


Figura 2

Una vez iniciado el asistente, lo primero que deberá indicarse es el nombre del proyecto. Se deberán aceptar las condiciones para poder continuar con el proceso.

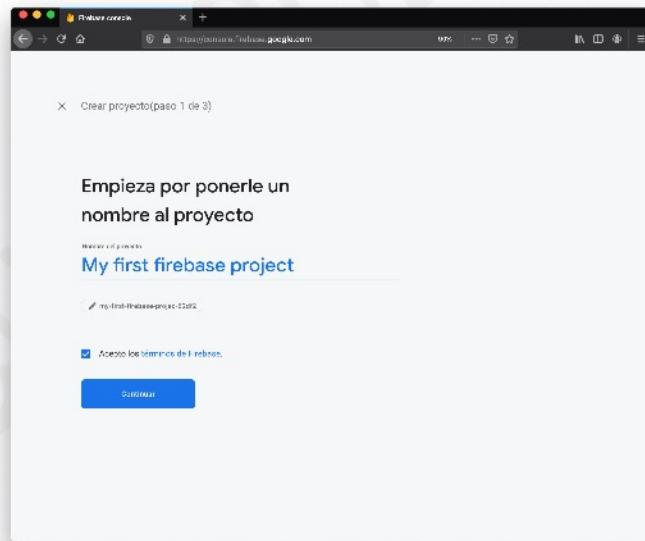


Figura 3

El siguiente paso será habilitar *Google Analytics* para poder analizar el uso de la aplicación.

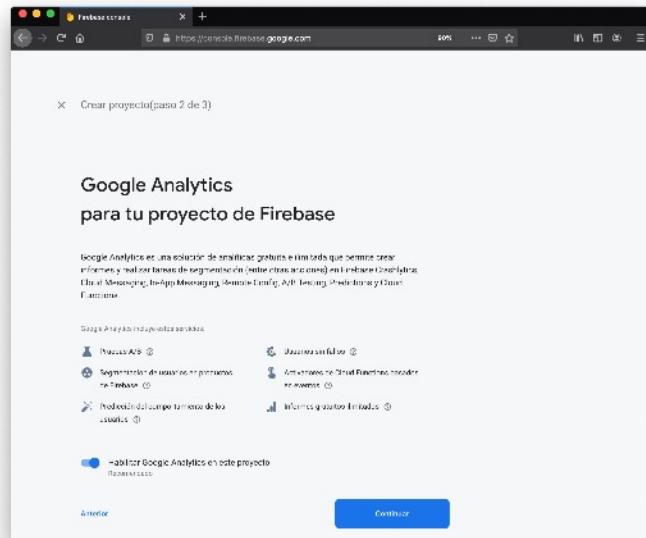


Figura 4

Al activar el uso de *Google Analytics* deberá seleccionarse una cuenta ya existente o, crear una nueva.

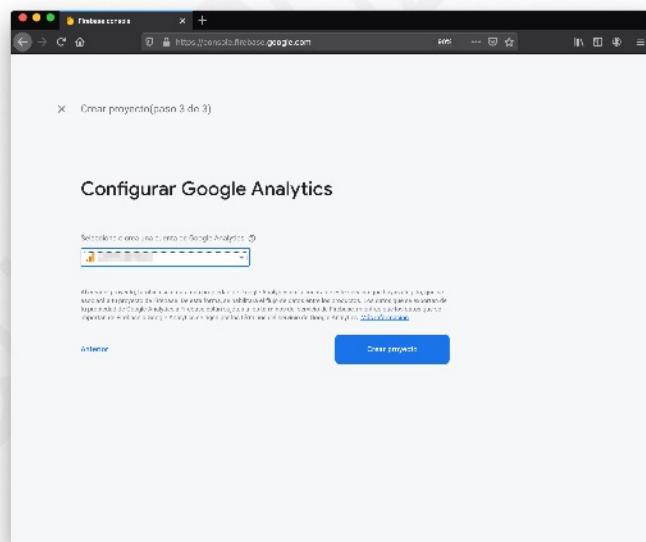
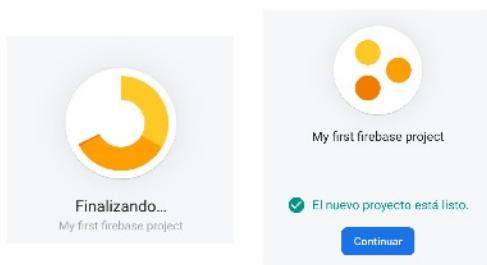


Figura 5

Tras pulsar el botón "Crear proyecto" se verá una barra de progreso indicando el estado de creación.

6 UNIDAD 10 FIREBASE



Una vez finalizada la creación, ya se podrá ir al panel de control pulsando "Continuar". La vista inicial del proyecto *Firebase* puede ser como se muestra en la siguiente imagen.

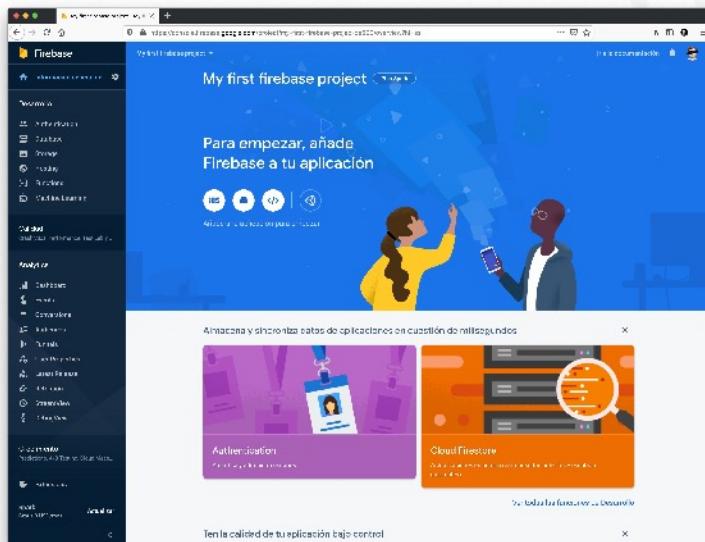
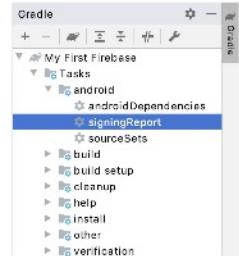


Figura 6

Una vez creado el proyecto en *Firebase*, deberá añadirse al proyecto creado en *Android Studio*. Previamente, se habrá creado un proyecto en **Android Studio**, el cual será el que se registre. Se seleccionará la opción para **Android** y seguirán los pasos indicados por el asistente.



El **nombre del paquete** debe ser el mismo que el del proyecto de **Android Studio**. El **apodo** es opcional, pero no está de más indicar uno para identificarlo en *Firebase*. Por último, el **certificado de firma**, también opcional, pero que si se quieren utilizar ciertas propiedades de Google, será necesario. Dicho certificado se puede obtener del certificado SHA-1 de *debug* generado automáticamente al instalar *Android Studio*². Una forma fácil de obtenerlo desde *Android Studio* es utilizando la opción **signingReport** de **Gradle**.



2 Obtener certificado SHA1 (<https://developers.google.com/android/guides/client-auth>)

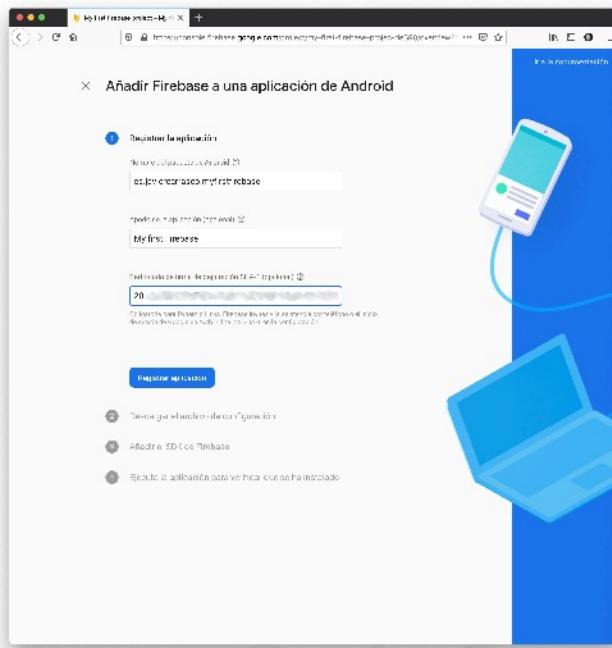


Figura 7

Una vez indicados los datos, ya se puede **Registrar aplicación**. Tras unos segundos, se podrá descargar el fichero `google-services.json` y seguir los pasos indicados en Android Studio.

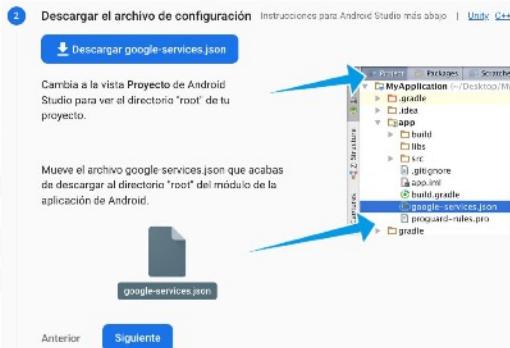


Figura 8

Una vez añadido el fichero al proyecto, puede pulsarse **Siguiente** y seguir las instrucciones que indica el siguiente paso. Añadir las dependencias en el *Gradle*.

Tras la sincronización del *Gradle*, si todo ha ido bien, pulsa **Siguiente** en *Firebase* para realizar el último paso, deberá lanzarse el proyecto de **Android Studio** en el emulador. Tras unos angustiosos segundos, si todo ha ido bien, se producirá el registro en *Firebase* y verás el siguiente mensaje en el navegador.

8 UNIDAD 10 FIREBASE

- Ejecuta la aplicación para verificar que se ha instalado

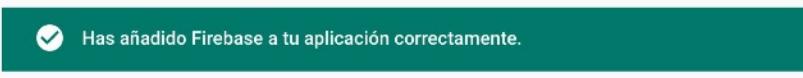


Figura 9

Ya puedes pulsar “Ir a la consola” y comprobar que ya tienes registrada la aplicación en el proyecto de *Firebase*.

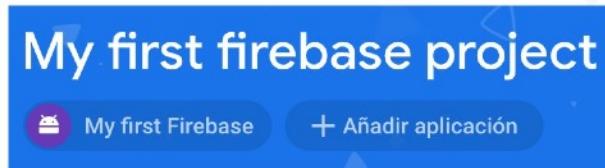


Figura 10

10.2. Crear una base de datos en *Firebase*

Se dispone de dos “tipos” de bases de datos en *Firebase*, **Realtime Database** y **Cloud Firestore**.

- **Realtime Database** es la primera base de datos que tuvo *Firebase*. Eficiente y con baja latencia para dispositivos móviles con una sincronización en tiempo real.
- **Cloud Firestore** de creación más reciente, aprovecha las ventajas de *Realtime Database* e incluye un modelo más intuitivo. También dispone de consultas más rápidas y eficaces.

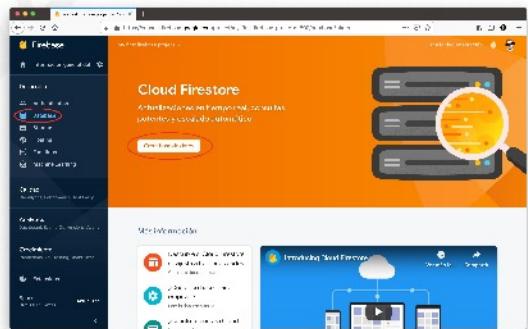


Figura 11

Según las necesidades requeridas para la aplicación, deberás inclinarte por un tipo de base de datos u otro³. Ambas disponen de gran facilidad para almacenar datos simples, pero, para datos complejos es mejor utilizar *Cloud Firestore*.

Para crear una base de datos asociada al proyecto, deberás seleccionar la opción *Database* de la parte izquierda en la consola de *Firebase* del proyecto.

Siguiendo los pasos del asistente, crearás una base de datos para el proyecto. El primer paso será elegir el modo, la diferencia radica en la seguridad.

3 Comparativa (<https://firebase.google.com/docs/database/rtdb-vs-firebase>)

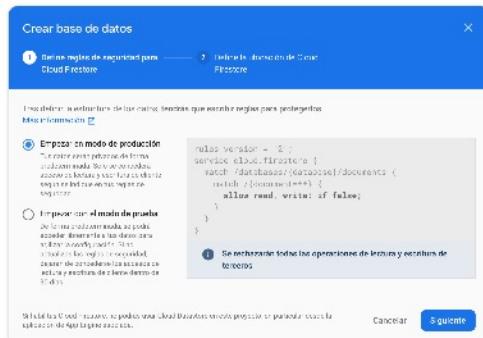


Figura 12

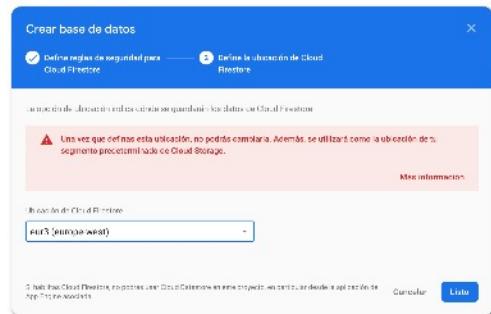


Figura 13

En el segundo paso se elegirá la ubicación de nuestra base de datos, y se habrá terminado la creación de la base de datos. Ahora, ya puedes seleccionar el tipo con el que trabajar.

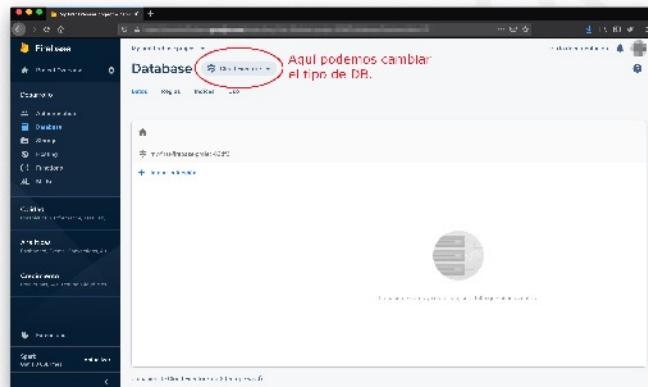


Figura 14

Realtime Database

Se comenzará por utilizar *Realtime Database*, como ya se ha comentado, ésta estructura los datos en formato JSON, lo cual requerirá una planificación clara y bien estructurada a la hora de trabajar con ella.

En primer lugar, como se trabajará en modo desarrollo, habrá que asegurarse que se dispone de acceso la base de datos. Como todavía no se ha activado ningún método para realizar la autenticación, se deben modificar las **reglas** de acceso.

Podrás ver un *script* para configurar las reglas, y un botón **Zona de prueba de reglas**, que permitirá comprobar la configuración. Para este caso, se deberán tener a *true* las opciones *read* y *write*.

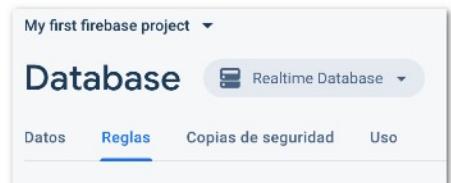


Figura 15

10 UNIDAD 10 FIREBASE

```
1  {
2      /* Visit https://firebase.google.com/docs/database/security to learn more about
3          security rules. */
4      "rules": {
5          ".read": true,
6          ".write": true
7      }
8 }
```

De vuelta a **datos**, se añadirá una estructura de datos para que pueda ser consultada desde la aplicación.



Figura 16

Para añadir datos, o nodos, basta con pulsar sobre el signo más que aparece junto al nodo ya existente. Se deberá ir añadiendo información mediante el sistema **clave-valor**. Si quiere añadirse un nodo, como *gato* de la imagen anterior, no se indicará el valor.

Con esta estructura de datos, ya se puede volver a **Android Studio** para que la aplicación pueda consultar esta información. Una modificación que deberá hacerse antes de continuar es añadir una nueva dependencia al *Gradle* de módulo.

```
1 implementation "com.google.firebaseio:firebase-database:20.0.0"
```

Esta librería permitirá hacer uso de la clase `FirebaseDatabase()`, necesaria para poder utilizar **Realtime Database**. Puedes consultar las bibliotecas⁴ disponibles en la web de *Firebase*. También será necesario solicitar el permiso para poder acceder a *Internet* en la aplicación.

Ya en la clase principal, se deberá crear la referencia a la base de datos de *Firebase*. Sería como establecer un puntero a la raíz de la propia base de datos, la cual debe verse como una estructura de árbol por la que se irá navegando. Esta referencia se recogerá de la siguiente forma.

```
1 val database = FirebaseDatabase.getInstance().reference
```

Si se dispone de más de una base de datos en el mismo proyecto *Firebase* (de pago), debería especificarse la URL a la hora de obtener la instancia.

```
1 val database = FirebaseDatabase
2     .getInstance("https://my-.....com/")
3     .reference
```

4 Bibliotecas disponibles en *Firebase* (<https://firebase.google.com/docs/android/setup?authuser=0#available-libraries>)

Una vez obtenido el punto de partida, se debe recoger, establecer o **leer**, la referencia al nombre y la raza del gato almacenado.

```

1 // Se obtiene la referencia al nombre del gato.
2 val dbfNombre = database.child("mascotas")
3     .child("gato")
4     .child("nombre")

```

Fíjate como se va pasando de hijo en hijo hasta llegar al dato en cuestión, de ahí la necesidad de planificar la estructura a la hora de utilizar **Realtime Database**.

Ahora ya puedes trabajar con los datos, o más bien con las referencias obtenidas. Esto se llevará a cabo de la siguiente forma.

```

1 /**
2 * Se crea un listener para que sean notificados los cambios
3 * en el nombre.
4 */
5 dbfNombre.addValueEventListener(object : ValueEventListener {
6     // Se llama cuando se cancela la lectura, o se produce un error.
7     override fun onCancelled(error: DatabaseError) {
8         Log.e("onCancelled", "Error!", error.toException())
9     }
10
11     // Se ejecuta cuando se obtiene el valor.
12     override fun onDataChange(snapshot: DataSnapshot) {
13         binding.tvNombre.text = "Nombre: ${snapshot.value}"
14     }
15 })

```

Al crear el *listener* deben sobrecargarse dos métodos pertenecientes a `ValueEventListener`, `onCancelled()` para controlar posibles errores durante la recuperación del dato, y `onDataChange()` para indicar que cuando se obtiene el valor.

El *listener* `addValueEventListener()` se encontrará en "escucha" continua de la referencia establecida, de manera que si se produce algún cambio en el dato, se notificará dicho cambio a la referencia. Puedes hacer la prueba lanzando la aplicación y cambiar el dato desde la consola de *Firebase*, viendo como se produce la actualización inmediatamente en la aplicación.

Es posible que en ocasiones, según la naturaleza de la aplicación que se esté desarrollando, no sea necesario tener una escucha continua a la referencia de algún dato. Si se diese el caso, en lugar de crear este *listener* continuo, se utilizaría `addListenerForSingleValueEvent()`. Su funcionamiento es exactamente igual al visto en el método `addValueEventListener()`, simplemente habría que cambiar un evento por otro, pero este se ejecutará únicamente una vez o cada vez que sea llamado de forma manual.

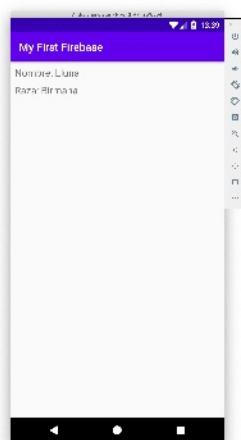


Figura 17

Ejercicios propuestos

10.1. Establece la referencia para la raza del gato y su *listener* `addValueEventListener()` para que se muestre en un *TextView* bajo el nombre del gato.

A continuación, se verá como **escribir** en la base de datos, se empezará por crear un botón en la UI que permita lanzar la escritura. Como se verá en el código siguiente, deberá localizarse la referencia (raíz relativa) a partir de la cual se quiere escribir. Una vez ubicados, se utilizará el método `setValue()`.

```

1 // Añadir usuarios a Firebase.
2 binding.btnAddUsers.setOnClickListener {
3     val users: MutableList<User> = ArrayList()
4
5     users.add(User("Javier", "Carrasco"))
6     users.add(User("Nacho", "Cabanes"))
7     users.add(User("Patricia", "Aracil"))
8     users.add(User("Juan", "Palomo"))
9     users.add(User("Raquel", "Sánchez"))
10
11     database.child("usuarios").setValue(users)
12 }
```

En este caso se está utilizando una *MutableList* para crear un conjunto de datos mediante un *data class* (nombre y apellido), esto permitirá insertar varios nodos a la vez manteniendo una misma estructura.

```

1 data class User(
2     var name: String? = "",
3     var surname: String? = ""
4 )
```

El resultado que se obtendría en *Firebase* es el que muestra la figura adjunta, además, fíjate que se creará automáticamente un índice.

Si se pulsa 20 veces el botón, el resultado será el mismo, no se duplicarán, simplemente reescribe desde la raíz. Por ejemplo, si añadimos justo a continuación de la instrucción `setValue(users)` las dos siguientes líneas:

```

1 database.child("usuarios").setValue("Pedro")
2 database.child("usuarios").setValue("Candy")
```

El resultado que se obtendría sería el que muestra la figura mostrada.



Figura 19

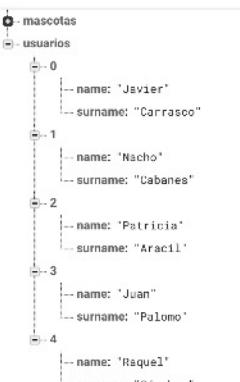


Figura 18

Motivo por el cual hay que planificar muy bien como quiere guardarse la información en **Realtime Database**. Y muy importante, no actualiza los datos existentes, elimina y vuelve a crear la información. Si se quiere **actualizar** datos de un nodo deberá utilizarse el método `updateChildren()`. Este método necesitará como parámetro un *HashMap*, indicando la clave a modificar y los datos nuevos.

```

1 // Actualizar usuario.
2 binding.btnUpdateUser.setOnClickListener {
3     val userUpdate = HashMap<String, Any>()
4     userUpdate["0"] = User("Javi", "Hernández")
5
6     database.child("usuarios").updateChildren(userUpdate)
7 }
```

Como ya se ha comentado, es necesario tener un conocimiento total de la estructura almacenada en **Realtime Database** de Firebase para saber en todo momento a que se está accediendo.

Si lo que se quiere es **eliminar** algún nodo, se necesitará nuevamente obtener la referencia completa al nodo que se desee eliminar.

```

1 // Eliminar usuario.
2 binding.btnDeleteUser.setOnClickListener {
3     database.child("usuarios")
4         .child("0")
5         .removeValue()
6 }
```

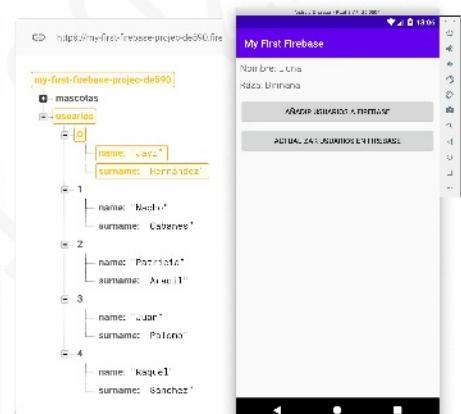


Figura 20

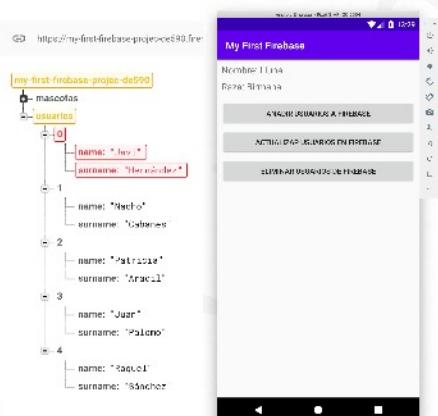


Figura 21

Dos métodos que pueden resultar útiles para trabajar con **Realtime Database**, evidentemente existen muchos más y puedes consultarlos en la documentación de **Firebase**, son los siguientes:

- **push()**, junto con `key`, generará una clave aleatoria (`database.child("usuarios").push().key`) para utilizarla en la creación de un nuevo nodo.
- **parent**, devuelve el nodo padre (`database.child("usuarios").parent`) de la referencia apuntada en formato URI.

Ejercicios propuestos

10.2. Crea una aplicación utilizando **Android Studio + Kotlin** con API 21 + **Realtime Database** (*Firebase*) que simule una lista de la compra, salvando las distancias. La aplicación deberá contener un campo de texto para poder escribir el producto (tomates, lechuga, etc) y un botón para añadir a la lista.

Para no complicar, la lista en cuestión deberá aparecer en un *TextView* según vayan añadiéndose artículos.

La aplicación también deberá contener un botón para eliminar artículos, con la siguiente particularidad, borrará siempre el último elemento que aparezca en la lista.

Cloud Firestore

Este modelo de datos es una base de datos **NoSQL orientada a documentos**. En vez de utilizar tablas y filas, se utilizarán colecciones y documentos.

Cada uno de los documentos contendrá un conjunto de elementos *clave-valor*, y todos los documentos deberán estar almacenados en colecciones.

Además, los documentos podrán contener sub-colecciones y otros objetos, en última instancia, contendrán campos primitivos (cadenas de texto, enteros, etc).

La creación de los documentos se hará de forma implícita, únicamente se deberán asignar los datos, si el documento no existe se crea.

Comienza creando un nuevo proyecto en **Android Studio** y añadiendo una nueva aplicación a la consola de *Firebase*, en el proyecto creado anteriormente en la opción *Settings*. Una vez añadida, vuelve a la opción **Database**, y esta vez, se trabajará con **Cloud Firestore**.

Como se hizo con *Realtime Database*, deberán ajustarse las reglas para permitir el acceso, ya que todavía no se ha activado la autenticación. En la sección **Reglas** deberás ajustar el *script* para que quede como se muestra a continuación. Además, también se dispone de un simulador para comprobar la configuración.

```

1 rules_version = '2';
2 service cloud.firestore {
3     match /databases/{database}/documents {
4         match /{document=**} {
5             allow read, write: if true;
6         }
7     }
8 }
```

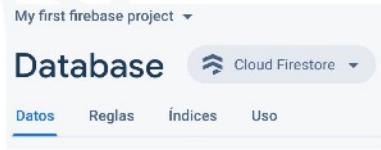


Figura 22

De vuelta a la sección **Datos** de la base de datos de *Cloud Firestore*, se creará la colección "profesores" pulsando sobre la opción **Iniciar colección** y se añadirán una serie de documentos.

Cuando se crea una colección por primera vez en *Cloud Firestore*, pedirá que se cree el primer documento, esto es debido a que una colección sin documentos no tiene razón de ser, por lo que no sería creada.

Campo	Tipo	Valor
nombre	string	Javier
apellido	string	Carrasco
modulo	string	PMDM

Figura 24

Una colección es un contenedor de documentos que combinan datos.

Ejemplo: La colección "usuarios" contendrá un documento para cada usuario.

Figura 23

Los datos de muestra para realizar este ejemplo serán los siguientes, obteniendo una colección con seis documentos en total:

ID, nombre, apellido, modulo
1, Javier, Carrasco, PMDM
2, Nacho, Cabanes, PMDM
3, Antonio, Rodríguez, DI
4, Lorena, López, ED
5, Fernando, Albert, RL
6, Jana, Taboada, SOM

Ya de vuelta a *Android Studio*, lo primero que se deberá hacer en la aplicación será añadir la biblioteca⁵ correspondiente para hacer uso de *Cloud Firestore* en el *gradle* a nivel de *app*.

```
1 implementation 'com.google.firebaseio:firebase-firebase:23.0.2'
```

Si has trabajado con este tipo de bases de datos, las instrucciones que vas a ver a continuación te sonarán mucho. Al igual que con *Realtime Database*, lo primero que se deberá hacer será obtener la instancia a la base de datos.

```
1 val db: FirebaseFirestore = FirebaseFirestore.getInstance()
```

Seguidamente se recogerá la colección con la que se desea trabajar, en este caso "profesores".

```
1 val profesCollection: CollectionReference = db.collection("profesores")
```

Observa como obtener un único documento de una colección, evidentemente, deberá conocerse el identificador del documento. En este caso se recogerá el documento con el identificador 1.

```
1 // Obtener un documento concreto.
2 val docRef: DocumentReference = profesCollection.document("1")
```

5 Bibliotecas disponibles (<https://firebase.google.com/docs/android/setup?authuser=0#available-libraries>)

16 UNIDAD 10 FIREBASE

El siguiente paso será añadir los *listeners* que permitan recoger la información del documento. Debes saber, que todas estas lecturas serán *cacheadas* (*Snapshot*), por lo que ante un posible fallo, se mostrará la información almacenada.

```
1 docRef.get().apply {
2     // Obtiene información, se lanza sin llegar a terminar la conexión.
3     addOnSuccessListener {
4         Log.d("addOnSuccessListener", "Cached document data: ${it.data}")
5
6         val texto = it["modulo"].toString() + " - " +
7             it["nombre"].toString() + " " +
8             it["apellido"].toString()
9         binding.tvShowData.text = texto
10    }
11
12    // Fallo de lectura.
13    addOnFailureListener { exception ->
14        Log.d("addOnFailureListener", "Fallo de lectura ", exception)
15    }
16 }
```

Como se puede ver, en primer lugar se ejecutará un `get()` sobre el documento para obtenerlo. Una vez hecho, mediante la estructura `apply` (opcional) se instancia el método `addOnSuccessListener()` para indicar que debe hacerse durante una lectura correcta y el método `addOnFailureListener()` para indicar que hacerse en caso de producirse un fallo. Esta estructura te recordará a un *try-catch*, fíjate que en caso de fallo se dispone de un parámetro `exception`.



Figura 25

Otra forma de leer y realizar acciones, pero esta vez una vez finalizada la conexión, sería la que se muestra a continuación. Éste sistema puede ser útil cuando se tiene que leer mucha información de la nube y no se quiere mostrar, u operar con ella, hasta que esté toda disponible.

```
1 docRef.get().apply {
2     addOnCompleteListener {
3         if (it.isSuccessful) {
4             // Documento encontrado en la caché offline.
5             val document = it.result
6
7             Log.d("addOnCompleteListener", "Document data: ${document?.data}")
8             val texto = document!!["modulo"].toString() + " - " +
9                 document["nombre"].toString() + " " +
10                document["apellido"].toString()
11            binding.tvShowData.text = texto
12        } else {
13            Log.d("addOnCompleteListener", "Fallo de lectura ", it.exception)
14        }
15    }
16 }
```

En este caso, deben gestionarse los posibles fallos, de ahí la comprobación `isSuccessful` al iniciar las acciones. El método `addOnCompleteListener()` tiene como parámetro una `Task<DocumentSnapshot!>` representada en la variable `it`, `Task` es una clase que representa una operación asíncrona, de ahí que se deba recuperar el documento en la línea `val document = it.result` para poder trabajar.

Pero no se dispone de **escucha activa** en estos métodos, si haces la prueba de modificar algún dato del documento, no verás una actualización automática en la aplicación. Si se necesita la escucha activa se optará por el siguiente método, dejando de lado `docRef.get()`.

```

1 // Escucha del documento, contrasta la caché con la base de datos.
2 docRef.addSnapshotListener { document, e ->
3     // Se comprueba si hay fallo.
4     if (e != null) {
5         Log.w("addSnapshotListener", "Escucha fallida!", e)
6         return@addSnapshotListener
7     }
8
9     if (document != null && document.exists()) {
10        Log.d("addSnapshotListener", "Información actual: ${document.data}")
11        val texto = document["modulo"].toString() + " - " +
12            document["nombre"].toString() + " " +
13            document["apellido"].toString()
14        binding.tvShowData.text = texto
15    } else {
16        Log.d("addSnapshotListener", "Información actual: null")
17    }
18 }
```

Fíjate en la instrucción `return@addSnapshotListener`, esta la verás muy a menudo con el uso de *lambdas* y, sirve para especificar que función ha ejecutado el `return`. Se utiliza con frecuencia en funciones anidadas.

Observa ahora como se pueden recuperar **todos los documentos** de una colección desde una aplicación.

```

1 // Obtiene todos los documentos de una colección (sin escucha).
2 profesCollection.get().apply {
3     addOnSuccessListener {
4         for (document in it) {
5             Log.d("DOC", "${document.id} => ${document.data}")
6             binding.tvShowData.append(
7                 document!!["modulo"].toString() + " - " +
8                     document["nombre"].toString() + " " +
9                     document["apellido"].toString() + "\n"
10            )
11        }
12    }
13 }
```

18 UNIDAD 10 FIREBASE

```
14     addOnFailureListener { exception ->
15         Log.d(
16             "DOC",
17             "Error durante la recogida de documentos: ",
18             exception
19         )
20     }
21 }
```

El resultado que se obtendría sería como el que se muestra en la figura. Pero, si se modifican datos en la base de datos, con este sistema que no tiene escucha activa, no se verán las modificaciones de manera inmediata. Para recuperar todos los documentos utilizando la escucha activa se deberá utilizar el siguiente sistema.



Figura 26

```
1 // Obtiene todos los documentos de una colección (con escucha).
2 profesCollection.addSnapshotListener { querySnapshot, firestoreException ->
3     if (firestoreException != null) {
4         Log.w(
5             "addSnapshotListener",
6             "Escucha fallida!",
7             firestoreException
8         )
9         return@addSnapshotListener
10    }
11
12    binding.tvShowData.text = ""
13    for (document in querySnapshot!!) {
14        Log.d("DOC", "${document.id} => ${document.data}")
15        binding.tvShowData.append(
16            document!!["modulo"].toString() + " - " +
17            document["nombre"].toString() + " " +
18            document["apellido"].toString() + "\n"
19        )
20    }
21 }
```

Ahora bien, seguramente, no siempre se pretenda obtener todos los documentos de una colección. Para filtrar, se deberá utilizar utilizar el método `where()`, este tiene tres parámetros, el campo a filtrar, la operación de comparación y un valor. Pero, para **iOS, Android, Java y/o Kotlin**, el operador de comparación se nombra de forma explícita en el nombre del método.

```
1 // ==, <, <=, >, >=
2 .whereEqualTo("state", "CA")
3 .whereLessThan("population", 100000)
4 .whereLessThanOrEqualTo("name", "San Francisco")
5 .whereGreaterThanOrEqual("population", 100000)
6 .whereGreaterThanOrEqual("name", "San Francisco")
7 // Contenido en el Array.
8 .whereArrayContains("regions", "west_coast")
```

Si se quiere utilizar los métodos `where`, se deberá utilizar justo antes de ejecutar el método `get()` en el caso de no utilizar la escucha activa, y antes del método `addSnapshotListener()` en el caso de la escucha activa.

A continuación, duplica el `TextView` del ejemplo como `tvShowData2`, de forma que se puedan mostrar datos simultáneamente con y sin escucha, y añadiendo los métodos `where`.

```
1 // Obtiene todos los documentos de una colección (sin escucha).
2 profesCollection.whereEqualTo("modulo", "PMDM").get().apply {
3     addOnSuccessListener {
4         binding.tvShowData.text = "Sin escucha\n"
5         for (document in it) {
6             Log.d("DOC", "${document.id} => ${document.data}")
7             binding.tvShowData.append(
8                 document!!["modulo"].toString() + " - " +
9                     document["nombre"].toString() + " " +
10                    document["apellido"].toString() + "\n"
11            )
12        }
13    }
14    addOnFailureListener { exception ->
15        Log.d(
16            "DOC",
17            "Error durante la recogida de documentos: ",
18            exception
19        )
20    }
21 }
22
23 // Obtiene todos los documentos de una colección (con escucha).
24 profesCollection.whereEqualTo("modulo", "ED")
25     .addSnapshotListener { querySnapshot, firebaseFirestoreException ->
26         if (firebaseFirestoreException != null) {
27             Log.w(
28                 "addSnapshotListener",
29                 "Escucha fallida!.",
30                 firebaseFirestoreException
31             )
32             return@addSnapshotListener
33         }
34         binding.tvShowData2.text = "Con escucha\n"
35         for (document in querySnapshot!!) {
36             Log.d("DOC", "${document.id} => ${document.data}")
37             binding.tvShowData2.append(
38                 document!!["modulo"].toString() + " - " +
39                     document["nombre"].toString() + " " +
40                     document["apellido"].toString() + "\n"
41             )
42         }
43     }
```

20 UNIDAD 10 FIREBASE

Gracias al uso de los métodos `where` se conseguirá el resultado que muestra la figura.

Cloud Firestore tiene ciertas **limitaciones** a la hora de ejecutar consultas:

- Consultas con filtros de rango en diferentes campos, por ejemplo, `edad >= 18` y `saldo <=1000`.
- Consultas que utilicen el operador lógico OR. Para ello, se deberá crear una consulta independiente para cada condición OR y combinar los resultados de la consulta en la aplicación.
- Consultas con una cláusula `!=`. En tal caso, se dividirá la consulta en una de tipo "mayor que" y otra de tipo "menor que". Por ejemplo, si se busca aquellos cuya `edad` sea distinta de 30, se deberán combinar las consultas `edad < 30` y `edad > 30`.

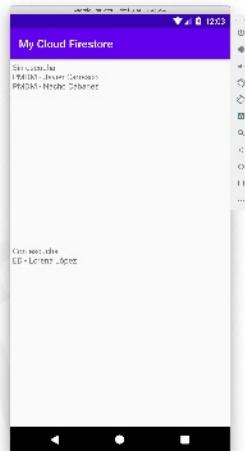


Figura 27

Si quieras, por ejemplo, obtener aquellos profesores de "PMDM" y que además se llamen "Javier", deberás enlazar los métodos `where`, para finalmente, añadir el `get()` o `addSnapshotListener()` según sea el caso.

```
1 profesCollection.whereEqualTo("nombre", "Javier")
2 .whereEqualTo("modulo", "PMDM")
```

A continuación, se verá como **añadir** documentos a *Cloud Firestore* desde la propia aplicación. Para simplificar, se añadirá un botón entre los dos `TextView` anteriores que permitirá añadir un documento a la colección.

Si quieras añadir, o modificar un documento ya existente, deberás utilizar el método `set()`, en realidad, crea o reemplaza el documento. Debes saber que al añadir un documento nuevo a *Cloud Firestore*, si no existe, lo crea, evidentemente, pero si ya existe, reemplazará los datos por los nuevos proporcionados.

A la hora de crear un documento, deberás tener en cuenta el identificador que quieras utilizar, lo más sencillo es dejar que la base de datos cree uno de manera aleatoria.

```
1 binding.btnAdd.setOnClickListener {
2     // Se crea la estructura del documento.
3     val profe = hashMapOf(
4         "nombre" to "Isabel",
5         "apellido" to "Díaz",
6         "modulo" to "FOL"
7     )
8
9     // Se añade el documento sin indicar ID, dejando que Firebase genere el ID
10    // al añadir el documento. Para esta acción se recomienda add().
11    profesCollection.document().set(profe)
12        // Respuesta si ha sido correcto.
```

```

13     .addOnSuccessListener {
14         Log.d("DOC_SET", "Documento añadido!")
15     }
16
17     // Respuesta si se produce un fallo.
18     .addOnFailureListener { e ->
19         Log.w("DOC_SET", "Error en la escritura", e)
20     }
21 }
```

El resultado de la pulsación del botón será un nuevo documento creado en la colección de *Cloud Firestore*. La pulsación sobre el botón añadir actualizará el *TextView* que contiene la escucha activa. El resultado en *Firebase* será el siguiente.

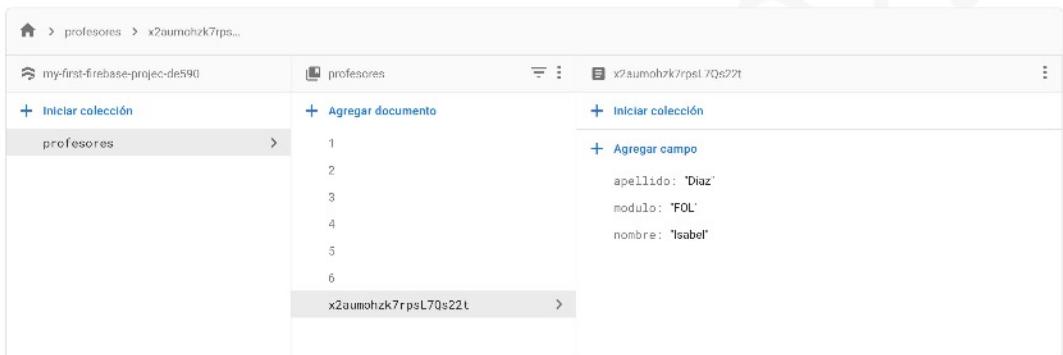


Figura 28

Si quieras especificar el *id*, deberás indicarlo, eso se hará en la instrucción `profesCollection.document(<tu ID Aquí>)`, pero como *Firebase* no crea *ids* autoincrementables, deberás buscar la forma de hacerlo. Por ejemplo, a mano, si sabes que el siguiente documento será el 7, puedes hacerlo a tiro fijo.

```

1     /**
2      * profesCollection.document("7")
3      *   .set(profe)
4     */
```

Problema, que al pulsar el botón "Añadir", la primera vez se creará el documento, pero de ahí en adelante, se actualizará constantemente. Verás como se añade el documento número 7 en la base de datos, pero en la aplicación, únicamente se verá como se añade el nuevo profesor en la primera pulsación, pero ya después no se apreciará ningún cambio. La otra opción, es crear una variable que controle el identificador, por ejemplo:

```

1     /**
2      * profesCollection.document(contadorId.toString())
3      *   .set(profe)
4     */
```

22 UNIDAD 10 FIREBASE

La variable `contadorId` se actualizará en el método `addSnapshotListener()` que se utiliza para completar el `TextView tvShowData2`, ya que al tener la escucha activa, está asegurada su actualización constante. Se hace uso del método `size()`, `contadorId = querySnapshot!!.size() + 1`. Pero este sistema, si la cantidad de usuarios es muy grande, tal vez no sea el más adecuado.

Volviendo la método `set()`, debes estar seguro de la existencia del documento a la hora de añadir uno nuevo, ya que su existencia puede hacer que sobrescribas datos de manera no deseada. También se recomienda añadir el identificador al documento siempre que se utilice el método `set()`.

```
1 val data = hashMapOf(  
2     "nombre" to "Javier",  
3     "apellido" to "Carrasco"  
4 )  
5 profesCollection.document("nuevo-profe").set(data)
```

Este código crearía un nuevo documento en *Cloud Firestore* en la colección *profesores* con el identificador *nuevo-profe*. Si por el contrario, el identificador de los documentos no es relevante, lo más recomendable es utilizar el método `add()` para la creación de documentos.

A continuación, se añadirá un documento utilizando el método `add()`, además se creará una nueva colección, "*modulos*".

```
1 binding.btnAdd.setOnClickListener {  
2     // Se selecciona la colección o la crea si no existe.  
3     val topicsCollection = db.collection("modulos")  
4  
5     val modulo = hashMapOf(  
6         "abreviatura" to "PMDM",  
7         "nombre" to "Programación Multimedia y Dispositivos Móviles"  
8     )  
9  
10    topicsCollection  
11        .add(modulo)  
12        .addOnSuccessListener {  
13            Log.d("DOC_ADD", "Documento añadido, id: ${it.id}")  
14        }  
15        .addOnFailureListener { e ->  
16            Log.w("DOC_ADD", "Error añadiendo el documento", e)  
17        }  
18 }
```

El *log* de **Android Studio** mostrará la siguiente línea al ejecutar la escritura si todo ha funcionado correctamente.

```
2020-09...co.mycloudfirestore D/DOC_ADD: Documento añadido, id: 0ib6nogqn435hxr26ijG
```

El resultado que se obtendría en la consola de *Firebase* sería algo parecido a lo que puedes ver en la siguiente figura.

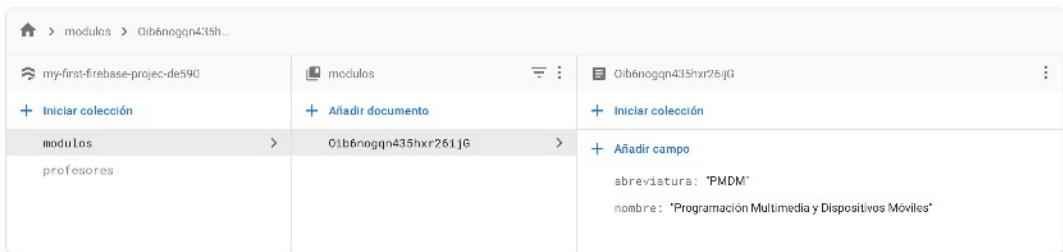


Figura 29

Otra manera de añadir documentos puede ser crear el documento, obtener su referencia y después añadir el resto de información. Esta operación deberá llevarse a cabo mediante el método `set()`, por ejemplo.

```

1 // Se crea un documento nuevo obteniendo su referencia.
2 val refNewTopic = db.collection("modulos").document()
3
4 val modulo = hashMapOf(
5     "abreviatura" to "PMDM",
6     "nombre" to "Programación Multimedia y Dispositivos Móviles"
7 )
8 // Se añaden los datos.
9 refNewTopic.set(modulo)

```

Si necesitas **actualizar**, o **modificar** campos, deberás hacer uso del método `update()`, éste permitirá realizar cambios sin modificar un documento por completo.

Para este tipo de operaciones es necesario conocer el identificador del documento a modificar, volviendo al documento creado anteriormente con el identificador automático `Oib6nogqn435hx26ijG`. Si se desea modificar el valor del campo `abreviatura`, deberá realizarse la siguiente operación.

```

1 // Se obtiene la referencia del documento a actualizar.
2 val refUpdTopic = db.collection("modulos").document("Oib6nogqn435hx26ijG")
3
4 refUpdTopic
5     .update("abreviatura", "MÓVILES")
6     .addOnSuccessListener {
7         Log.d("DOC_UPD", "Documento actualizado correctamente")
8     }
9     .addOnFailureListener { e ->
10        Log.w(
11            "DOC_UPD",
12            "Error al actualizar el documento",
13            e
14        )
15    }

```

Si se quiere actualizar varios campos a la vez, se utilizará `mapOf()` en el método `update()`.

```

1   ...
2   .update(mapOf(
3       "abreviatura" to "MÓVILES",
4       "nombre" to "Móviles con Kotlin"
5   ))
6   ...

```

Para **eliminar** documentos de *Cloud Firestore*, al igual que ocurre con la actualización, se debe obtener la referencia al documento que se desea borrar. El mecanismo será sencillo, se utilizará el método `delete()` y se añadirán los *listeners* para comprobar el resultado de la operación.

```

1 // Se obtiene la referencia del documento a eliminar.
2 val refDelTopic = db.collection("modulos").document("0ib6nogqn435hx26ijG")
3
4 refDelTopic
5     .delete()
6     .addOnSuccessListener {
7         Log.d("DOC_DEL", "Documento eliminado correctamente")
8     }
9     .addOnFailureListener { e ->
10        Log.w(
11            "DOC_DEL",
12            "Error al eliminar el documento",
13            e
14        )
15    }

```

Es importante saber que desde *Android* no es recomendable eliminar colecciones enteras, de hecho, la documentación oficial no muestra el proceso como medida de protección del propio *Firebase*. Si se quiere vaciar el contenido de una colección, se eliminará documento a documento.

10.3. Tipos compuestos en Firebase

En *Cloud Firestore* se dispone de dos tipos de datos que pueden contener campos anidados, los **mapas** y los **arrays**. A continuación, se verá como crear este tipo de campos y como acceder a ellos.

Los mapas se encontrarán representados como puede verse a continuación, mediante el sistema de *clave-valor*:

```
datos: {capital: "Sevilla", habitantes: 709000}
```

Los arrays aparecerán representados en su forma típica habitual:

```
provincias: ["Almería", "Granada", "Córdoba", "Jaén", "Sevilla", "Málaga", "Cádiz",
 "Huelva"]
```

Desde la consola de *Firebase* la nueva colección y sus documentos como se muestra en la siguiente figura.

The screenshot shows the Firebase Firestore interface. On the left, there's a sidebar with a tree view of collections: 'my-first-firebase-project-de590' (selected), 'Comunidades' (selected), 'modulos', and 'profesores'. In the main area, under 'Comunidades', there are sub-collections: 'Andalucía', 'Aragón', and 'Comunidad Valenciana' (selected). Under 'Comunidad Valenciana', there are fields: 'capital' (with value 'Valencia'), 'habitantes' (with value '791413'), and 'provincias' (with a list of three items: 'Castellón', 'Valencia', and 'Alicante').

Figura 30

Destacar como *Firebase* muestra el índice de cada uno de los elementos que forman el *array* creado dentro del documento.

Para este nuevo ejemplo, se ha añadido una nueva aplicación al proyecto creado en *Firebase*, no es necesario crear un proyecto nuevo por cada nueva aplicación. Es posible tener varias *apps* que utilicen el mismo proyecto de *Firebase*. Para esto, deberás ir a **Settings > General** y en la sección **Tus aplicaciones**, añadir una nueva *app* siguiendo los pasos anteriormente vistos.

The screenshot shows the Firebase console's configuration screen for the project 'My first firebase project'. On the left, there's a sidebar with 'Descripción general del proyecto' and 'Desarrollo' selected. The main area has tabs: 'General' (selected), 'Cloud Messaging', and 'Integraciones'. Under 'General', there's a section for 'Apps para Android' with three entries: 'My Cloud Firestore 2' (with package name 'es.javiercarrasco.mycloudfirebase2'), 'My Cloud Firestore' (with package name 'es.javiercarrasco.mycloudfirebase'), and 'My first Firebase' (with package name 'es.javiercarrasco.myfirst.firebaseio'). To the right, there's a section for 'Configuración de la app' for 'My Cloud Firestore 2'. It includes a download link for 'google-services.json', a SHA-1 fingerprint ('1:4...'), a field for 'Sobrenombre de la app' ('My Cloud Firestore 2'), a 'Nombre del paquete' field ('es.javiercarrasco.mycloudfirebase2'), a 'Huellas digitales del certificado SHA' field with a SHA-1 value ('28:...'), and a button to 'Agregar huella digital'. At the bottom right, there's a 'Quitar esta app' button.

Figura 31

26 UNIDAD 10 FIREBASE

En este ejemplo se creará una nueva aplicación, *My Cloud Firestore 2*. Esta se limitará a crear documentos que incluyan este tipo de datos, mostrarlos y eliminar documentos. En la figura que aparece junto al texto puedes ver el aspecto que tendrá.

Como puedes observar en la figura, la aplicación dispondrá únicamente de dos botones, el primero permitirá crear una colección, el segundo servirá para eliminar los documentos. También dispondrá de un *TextView* en el que se mostrará el contenido de la colección. El código que hará funcionar esta aplicación puedes verlo a continuación.

Antes de comenzar, deberás añadir la dependencia necesaria para manipular *Cloud Firestore*, que será la siguiente:

```
1 implementation 'com.google.firebaseio:firebase-firebase:23.0.2'
```

Pero existe otra que podrías utilizar en su lugar que tiene mayor afinidad con Kotlin, permitiendo escribir código más refinado.

Figura 32

```
1 implementation 'com.google.firebaseio:firebase-firebase-ktx:23.0.2'
```

Puedes utilizar cualquiera de las dos librerías. Volviendo al código, en primer lugar se crearán las siguientes dos propiedades para la clase *MainActivity*.

```
1 private val db: FirebaseFirestore = FirebaseFirestore.getInstance()  
2 private val collComunidades = db.collection("Comunidades")
```

También se creará el siguiente método que ayude a eliminar los documentos. Recordad que no se puede, o debe, eliminar colecciones completas, por lo que se eliminará documento a documento.

```
1 // Borra el documento indicando el ID por parámetro.  
2 fun borraDocumento(id: String) {  
3     // Elimina documento a documento, para ello,  
4     // se necesita conocer su identificador.  
5     collComunidades.document(id)  
6         .delete()  
7         .addOnSuccessListener {  
8             Log.d("DOC_DEL", "Documento eliminado correctamente")  
9         }  
10        .addOnFailureListener { e ->  
11            Log.w("DOC_DEL", "Error al eliminar el documento", e)  
12        }  
13 }
```

En el método *onStart()* se añadirá el siguiente código.

```
1 // Se obtienen todos los documentos de la colección (con escucha)  
2 // y se rellena el TextView.  
3 collComunidades.addSnapshotListener { querySnapshot, e ->  
4     if (e != null) {  
5         Log.w("DOC_SHOW", "Escucha fallida!.", e)
```



```
6     return@addSnapshotListener
7 }
8
9 binding.tvData.text = ""
10 for (document in querySnapshot!!) {
11     Log.d("DOC_SHOW", "${document.id} => ${document.data}")
12     binding.tvData.append("${document["comunidad"]}\n" +
13         "\tCapital: ${document["datos.capital"]}\n" +
14         "\tHabs: ${document["datos.habitantes"]}\n" +
15         "\tProvincias: ${document["provincias"]}\n\n"
16     )
17 }
18 }
```

Lo siguiente será crear el código para el botón **Crear colección**.

```
1 // Botón para añadir documentos a la colección.
2 binding.btnAdd.setOnClickListener {
3     // Se prepara la estructura de datos.
4     val comunidades = listOf(
5         mapOf( // Comunidad 1
6             "comunidad" to "Andalucía",
7             "datos" to mapOf(
8                 "capital" to "Sevilla",
9                 "habitantes" to 709000
10            ),
11            "provincias" to listOf(
12                "Almería",
13                "Granada",
14                "Córdoba",
15                "Jaén",
16                "Sevilla",
17                "Málaga",
18                "Cádiz",
19                "Huelva"
20            )
21        ), mapOf( // Comunidad 2
22            "comunidad" to "Comunidad Valenciana",
23            "datos" to mapOf(
24                "capital" to "Valencia",
25                "habitantes" to 791413
26            ),
27            "provincias" to listOf(
28                "Castellón",
29                "Valencia",
30                "Alicante"
31            )
32        ), mapOf( // Comunidad 3
33            "comunidad" to "Aragón",
34            "datos" to mapOf(
35                "capital" to "Zaragoza",
36            )
37        )
38    )
39    val db = FirebaseFirestore.getInstance()
40    db.collection(comunidades[0]["comunidad"])
41        .add(communidad)
42        .addOnSuccessListener {
43            Toast.makeText(context, "Documento añadido", Toast.LENGTH_SHORT).show()
44        }
45    }
46 }
```

```

36         "habitantes" to 680895
37     ),
38     "provincias" to listOf(
39         "Huesca",
40         "Zaragoza",
41         "Teruel"
42     )
43 )
44 )
45
46 // Se añade documento a documento.
47 for (doc in comunidades) {
48     Log.d("DOC_ADD", "Añadiendo documento " + doc["comunidad"])
49     collComunidades.document(doc["comunidad"].toString()).set(doc)
50         .addOnSuccessListener {
51             Log.d("DOC_ADD", "Documento añadido correctamente")
52         }
53         .addOnFailureListener { e ->
54             Log.w("DOC_ADD", "Error al añadir el documento", e)
55         }
56     }
57 }
```

Ya por último, el botón **Borrar colección**, que al tener ya preparado el método `borraDocumento()`, quedará como se muestra a continuación.

```

1 // Botón para eliminar documentos de la colección.
2 binding.btnDel.setOnClickListener {
3     borraDocumento("Andalucía")
4     borraDocumento("Comunidad Valenciana")
5     borraDocumento("Aragón")
6 }
```

Es posible añadir un elemento a un `array`, obteniendo la referencia del documento que lo contiene. Fíjate en el siguiente código de ejemplo.

```

1 // Se obtiene la referencia al documento
2 val refComunidad = collComunidades.document("Aragón")
3 // Se añade un valor nuevo al array provincias.
4 refComunidad.update("provincias", FieldValue.arrayUnion("Otro valor"))
```

El uso de `arrayUnion()` permite añadir un nuevo elemento a un `array`, pero únicamente si dicho elemento no existe previamente.

Lo mismo podría hacerse para eliminar un elemento del `array`, una vez obtenida la referencia del documento que contiene el `array`, se utilizaría el método `arrayRemove()`.

```

1 // Se obtiene la referencia al documento
2 val refComunidad = collComunidades.document("Aragón")
3 // Se elimina el valor del array provincias.
4 refComunidad.update("provincias", FieldValue.arrayRemove("Teruel"))
```

Ejercicios propuestos

10.3. Crea una aplicación utilizando **Android Studio + Kotlin** con API 21 + **Cloud Firestore** (**Firebase**) que será una lista de la compra mejorada. En su única *activity* se permitirá añadir elementos a la lista. La información del elemento será su nombre, su cantidad y si esa cantidad son unidades, gramos, kilos, paquetes, etc. Cada elemento añadido será un documento de la colección.

10.4. Modifica la aplicación creada en el ejercicio **10.3.**, añadiendo un botón que abra una nueva actividad y muestre la lista completa, indicando por cada elemento su nombre, cantidad y medida utilizada. Muestra la lista utilizando un *ListView* o un *RecyclerView* según consideres.

10.5. Invierte el orden de las actividades creadas en el ejercicio **10.4.**, de forma que la primera *activity* en mostrarse sea la lista, y sea ésta la que contenga un botón para añadir nuevos artículos. La segunda *activity* deberá ser ahora la que permita añadir artículos. El usuario permanecerá en esta actividad añadiendo artículos a la lista hasta que se pulse el botón volver a la lista, la cual se verá actualizada.

10.6. Añade funcionalidad a la lista de artículos creada en el ejercicio anterior, de tal forma que, al hacer *click* sobre el artículo, éste se elimine. También puedes intentar que la acción se produzca al hacer una pulsación larga en lugar de un *click*.