Programación Multimedia y Dispositivos Móviles

UD 14. Firebase

Javier Carrasco

Curso 2024 / 2025

Firebase

14. Firebase

Firebase[1] 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.

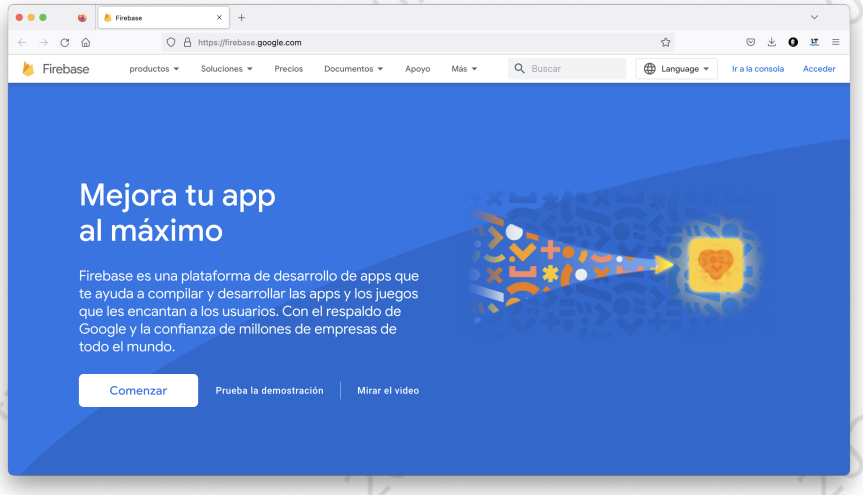


Figura 1

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

14.1. Primer proyecto Firebase

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

Una vez iniciada sesión, 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/)

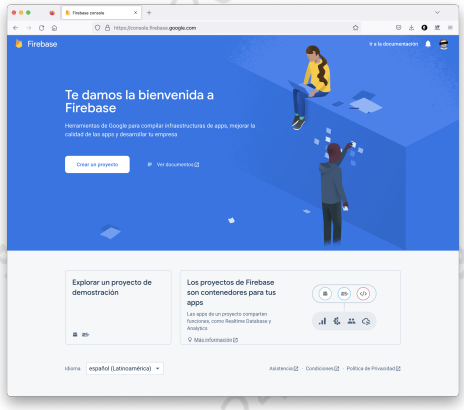


Figura 2

Una vez iniciado el asistente, deberá indicarse es el nombre del proyecto en primer lugar.

Figura 3

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

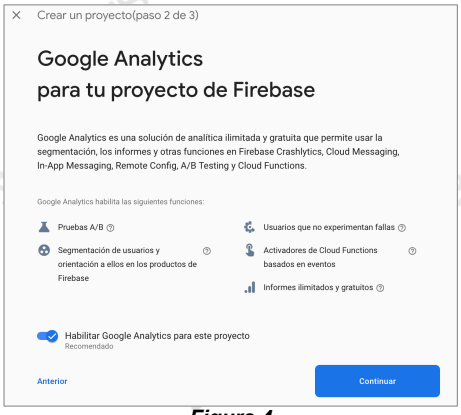


Figura 4

Al activar el uso de *Google Analytics* deberá seleccionarse una cuenta ya existente o, crear una nueva. Esto creará una vinculación entre *Google Analytics* y el proyecto Firebase que se está creando.

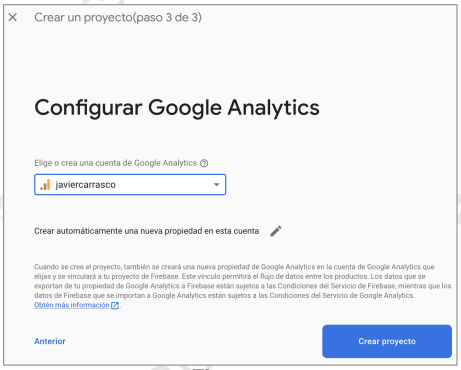


Figura 5

Tras elegir la cuenta (por defecto con la que iniciaste sesión) y pulsar el botón "Crear proyecto" se verá el progreso de creación del proyecto hasta que esté listo para su uso.

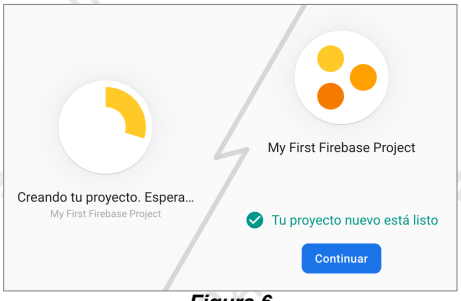
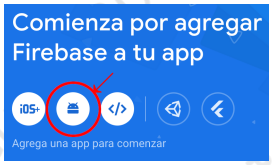


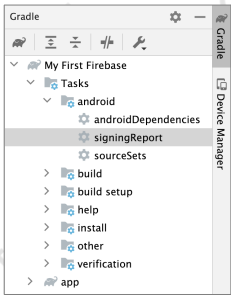
Figura 6

Una vez finalizada la creación, ya podrás acceder al panel de control pulsando "Continuar". La vista inicial de tu proyecto Firebase puede ser como se muestra en la siguiente figura.

Figura 7



Una vez funcionando el proyecto Firebase, deberá añadirse al proyecto previamente creado en **Android Studio**, el cual deberá registrarse en el proyecto Firebase. Se seleccionará la opción para **Android** y se seguirán los pasos indicados por el asistente.



El **nombre del paquete** debe ser el mismo que el del proyecto de **Android Studio**. El **apodo** (sobrenombre) 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**.

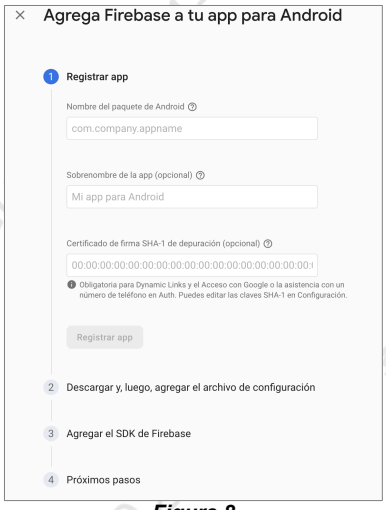


Figura 8

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 9

Una vez añadido el fichero al proyecto siguiendo las instrucciones que se muestran, puede pulsarse **Siguiente** y continuar con las instrucciones que se indican en el siguiente paso. Añadir las dependencias en el *Gradle*. En primer lugar, en el fichero ***build.gradle.kts*** a nivel de **proyecto** añadir:

```
plugins {
    ...

    // Add the dependency for the Google services Gradle plugin
    id("com.google.gms.google-services") version "4.4.0" apply false
}
```

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

Después, en el fichero ***build.gradle.kts*** a nivel de ***app*** se añadirán las siguientes dependencias.

```
plugins {
    ...

    // Add the Google services Gradle plugin
    id("com.google.gms.google-services")
}
...

dependencies {
    ...

    // Import the Firebase BoM
    implementation(platform("com.google.firebase:firebase-bom:32.3.1"))

    // TODO: Add the dependencies for Firebase products you want to use
    // When using the BoM, don't specify versions in Firebase dependencies
    implementation("com.google.firebase:firebase-analytics-ktx")
}
```

Tras la sincronizar el *Gradle*, si todo ha ido bien, pulsa **Siguiente** en Firebase para realizar el último paso. Ejecuta el proyecto de **Android Studio** para comprobar que se ejecuta correctamente, busca las siguientes dos líneas en la pestaña *Run* o *Logcat (package:mine tag:firebase)*.

```
... Device unlocked: initializing all Firebase APIs for app [DEFAULT]
... FirebaseApp initialization successful
```

Tras esto ya podrás ir a la consola de Firebase. Recuerda también añadir el permiso para Internet en el *manifest*.

En la pantalla principal de la consola podrás comprobar si ya tienes registrada la aplicación en el proyecto de Firebase.

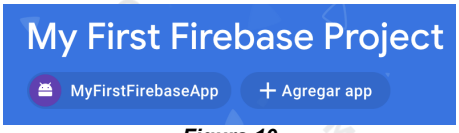


Figura 10

14.2. Crear una base de datos en Firebase

Existen 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 sincronización en tiempo real.
- **Cloud Firestore** llegó después y aprovecha las ventajas de *Realtime Database* incluyendo un modelo más intuitivo. También dispone de consultas más rápidas y eficaces.

Según las necesidades requeridas para la aplicación, deberás inclinarte por un tipo de base de datos u otro[3]. 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 *Compilación* de la parte izquierda en la consola de Firebase del proyecto.

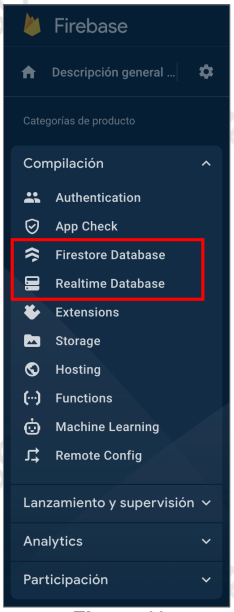


Figura 11

14.2.1. Realtime Database

Se comenzará por utilizar Realtime Database, como 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.

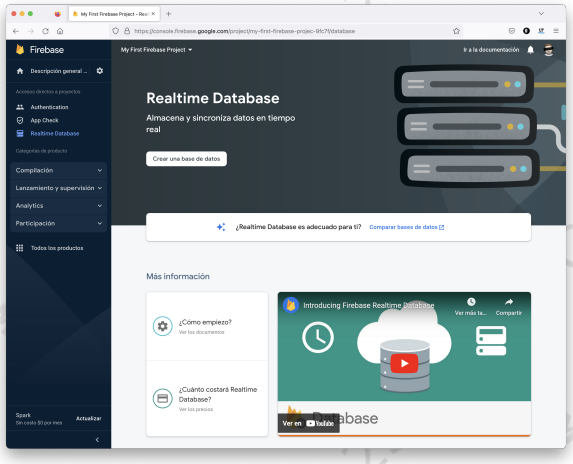


Figura 12

Al seleccionar la opción “Crear una base de datos” desde Realtime Database se lanzará un asistente. El primer paso será seleccionar la ubicación, por cercanía *“Bélgia (europe-west1)”*.

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

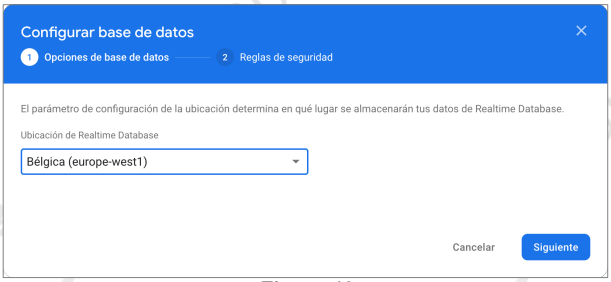


Figura 13

El siguiente paso sería establecer las reglas de acceso, de momento se dejarán en **modo de prueba**, acceso completo durante 30 días.

Figura 14

Otra opción es elegir el **modo bloqueado** y posteriormente modificar las reglas[4] a...

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

... indicando las opciones *read* y *write* a *true*, algo que puede ser peligroso si dejan así en producción. Esta modificación puedes hacerla en la pestaña **Reglas** de la consola de la base de datos creada.

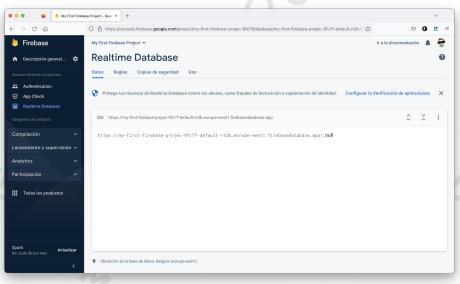


Figura 15

Tras habilitar la configuración se abrirá la vista inicial de tu base de datos Realtime Database.

En la pestaña **Datos** se añadirá una estructura de datos para que pueda ser consultada desde la aplicación para comenzar.

4 Reglas de seguridad básicas (https://firebase.google.com/docs/rules/basics)

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 "mascotas" o "gato" como se muestra en la figura, no se indicará el valor y se pulsará el signo más (+) a su altura.

Figura 17

La idea es conseguir una estructura inicial como la que se muestra a continuación.

Figura 18

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(Module:app)*.

```
implementation("com.google.firebase:firebase-database:20.2.2")
```

Esta librería permitirá hacer uso de la clase `FirebaseDatabase()`, necesaria para poder utilizar **Realtime Database**. Puedes consultar las bibliotecas[5] 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. Ésta referencia se recogerá haciendo uso de la clase `FirebaseDatabase` de la siguiente forma.

```
val database: DatabaseReference = FirebaseDatabase.getInstance().reference
```

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

Si se dispone de más de una base de datos en el mismo proyecto Firebase (de pago), o aparecen errores de conexión del tipo fallo de región, se deberá especificar la URL a la hora de obtener la instancia.

```
val database = FirebaseDatabase
    .getInstance("https://my-...database.app/")
    .reference
```

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

```
// Referencia al nodo "mascotas".
val dbfMascotas = database.child("mascotas")
// Referencia al nodo "gato".
val dbfGato = dbfMascotas.child("gato")
val dbfNombre = dbfGato.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.

```
// Se crea un listener para que sean notificados los cambios en el nombre.
dbfNombre.addValueEventListener(object : ValueEventListener {
    override fun onDataChange(snapshot: DataSnapshot) {
        binding.tvNombre.text = getString(R.string.txt_nombre, snapshot.value)
    }

    // Se llama cuando se cancela la lectura, o se produce un error.
    override fun onCancelled(error: DatabaseError) {
        Log.e("MainActivity", "Error al leer el valor de nombre.")
    }
})
```

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 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.

En ocasiones, según el tipo de aplicación que se esté desarrollando, no es necesario tener una "escucha" continua a la referencia de los datos. En tal caso, en vez de crear este *listener* continuo, se utilizará `addListenerForSingleValueEvent()`. Funciona exactamente igual al visto en el método *addValueEventListener()*, solo habría que cambiar un evento por otro, pero este se ejecutará una sola vez, o cada vez que sea llamado de forma manual.

El *listener* anterior se ha establecido sobre el nombre del gato, pero si quieres que se aplique a todos los datos de la mascota, bastará con modificar la variable `dbfNombre` por `dbfGato`, dejando la referencia en "gato". De esta forma se tiene acceso a las dos propiedades desde el *listener*.

```
binding.tvNombre.text = getString(R.string.txt_nombre, snapshot.child("nombre").value)
binding.tvRaza.text = getString(R.string.txt_raza, snapshot.child("raza").value)
```

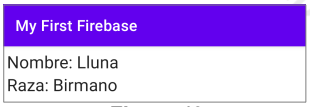


Figura 19

A continuación, se verá como **escribir** en la base de datos. Se empezará por añadir un botón a la UI para lanzar el proceso de 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()`.

```
// Añadir usuarios a Firebase.
binding.btnAddUsers.setOnClickListener {
    val users: MutableList<User> = mutableListOf()

    users.add(User("Javier", "Carrasco"))
    users.add(User("Nacho", "Cabanes"))
    users.add(User("Patricia", "Aracil"))
    users.add(User("Juan", "Palomo"))
    users.add(User("Raquel", "Sánchez"))

    database.child("ususarios").setValue(users)
}
```

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.

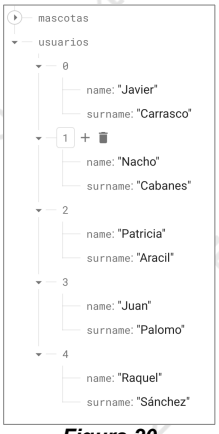


Figura 20

```
data class User(val name: String = "", val surname: String = "")
```

El resultado que se obtendría en Firebase es el mostrado en 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 se añade justo a continuación de la instrucción `setValue(users)` las dos siguientes líneas.

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

El resultado que se obtiene es el que muestra la figura, no se guarda el dato “Pedro”, guardando únicamente “Candy” y, quedando fuera de la estructura creada con la clase *User*.

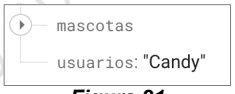


Figura 21

Motivo por el cual hay que planificar muy bien como quiere guardarse la información en **Realtime Database**. Otro dato importante, no se actualizan los datos existentes, sino que, 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 nuevos datos.

```
// Actualizar usuario.
binding.btnUpdateUsers.setOnClickListener {
    val userUpdate = HashMap<String, Any>()
    userUpdate["0"] = User("Javi", "Hernández")

    database.child("usuarios").updateChildren(userUpdate)
}
```

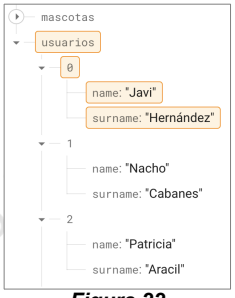


Figura 22

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.

```
// Eliminar usuario.
binding.btnDelUser.setOnClickListener {
    database.child("usuarios")
        .child("0")
        .removeValue()
}
```

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:

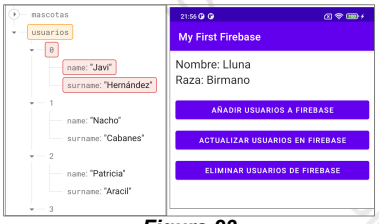


Figura 23

- **push()**, junto con *key*, se 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.

Ahora que se dispone de una cantidad de datos insertados, es posible que sea necesario poder recuperar más de uno a la vez. Para poder hacer esto, se ubicará la referencia en el nodo que contenga todos lo hijos que se desea obtener, volviendo al ejemplo de usuarios, sería el nodo “usuarios”.

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    // Se obtiene la referencia a los usuarios.
    val dbfUsers = database.child("usuarios")
```

Una vez obtenida la referencia, se añadirá el *listener* `addValueEventListener()` que se encargará de recoger todos los hijos.

```
dbfUsers.addValueEventListener(object : ValueEventListener {
    override fun onDataChange(snapshot: DataSnapshot) {
        val users: MutableList<User> = mutableListOf()

        binding.tvUsersList.text = null
        for (userSnapshot in snapshot.children) {
            users.add(userSnapshot.getValue(User::class.java)!!)

            binding.tvUsersList.append(
                "${userSnapshot.getValue(User::class.java)!!.name} " +
                        "${userSnapshot.getValue(User::class.java)!!.surname}\n"
            )
        }
    }

    override fun onCancelled(error: DatabaseError) {
        Log.e("onCancelled", "Error!", error.toException())
    }
})
```

Si observas el código del método `onDataChange()`, verás que además de mostrar los datos, se crea una *MutableList* de tipo *User* (*data class* creada anteriormente), con los datos obtenidos para poder trabajar con ellos posteriormente.

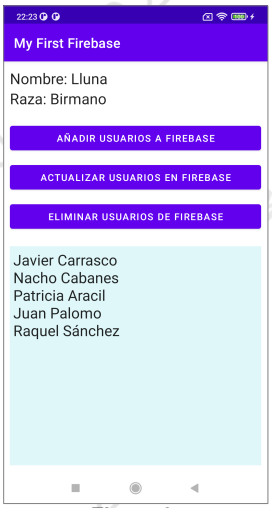


Figura 24

14.2.2. 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.

Figura 25

Comienza creando un nuevo proyecto en **Android Studio** (*MyCloudFirestore*, por ejemplo) y añadiendo una nueva aplicación a la consola de Firebase, en el proyecto creado anteriormente en la pestaña "General" de la "Configuración del proyecto". Recuerda añadir las dependencias para la configuración del proyecto en Android Studio. Una vez registrada la aplicación y configurada correctamente, en la consola de administración del proyecto selecciona la opción **Compilación**, y esta vez selecciona la opción **Firestore Database**, desde donde crearás una nueva base de datos **Cloud Firestore**.

Como se hizo con Realtime Database, deberán ajustarse las reglas de 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. En el primer paso del asistente esta vez se deberá indicar la seguridad, para realizar las pruebas y no tener problemas de acceso se dejará en **modo producción**. Asegúrate de cambiar la opción de ***if false*** por ***if true*** en la pestaña Reglas.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if true;  // Debe ser if true, no if false
    }
  }
}
```

El siguiente paso será indicar la ubicación, como avisa el asistente, una vez seleccionado no se podrá cambiar. Selecciona la opción que más se acomode a tu ubicación (en mi caso ***eur3***). Una vez habilitada, tras unos segundos, ya estarás en disposición de hacer uso de tu nueva base de datos Cloud Firestore.

Figura 26

Ya en 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.

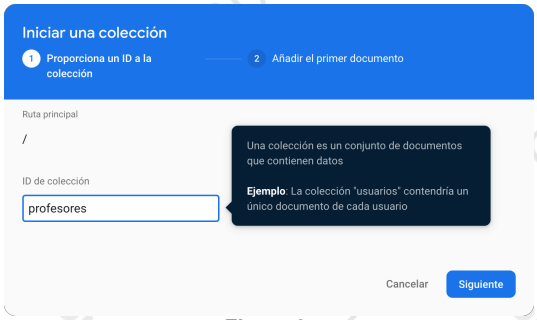


Figura 27

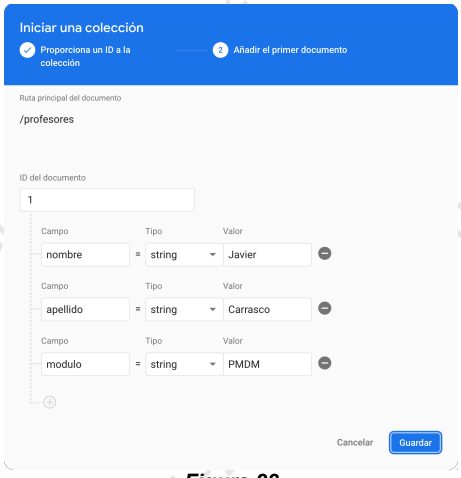


Figura 28

Los datos de muestra para realizar este ejemplo serán los siguientes, obteniendo una colección con seis documentos en total. Cada una de las líneas será un nuevo documento en la colección (**+Agregar documento**).

```
ID, nombre, apellido, modulo
----------------------------
1, Javier, Carrasco, PMDM
2, Nacho, Cabanes, Programación
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 biblioteca[6] correspondiente para hacer uso de **Cloud Firestore** en el *gradle(Module:app)*.

```
implementation("com.google.firebase:firebase-firestore:24.8.1")
```

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.

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

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

```
val colProfesores: 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, el primero que se ha añadido.

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

```
val docRef: DocumentReference = colProfesores.document("1")
```

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

```
docRef.get().apply {
    // Obtiene información, se lanza sin llegar a terminar la conexión.
    addOnSuccessListener {
        Log.d("addOnSuccessListener", "Cached document data: ${it.data}")

        val texto = "${it["modulo"]} - ${it["nombre"]} ${it["apellido"]}"
        binding.textView.text = texto
    }

    // Fallo de lectura.
    addOnFailureListener { e ->
        Log.d("addOnFailureListener", "Fallo de lectura ", e)
    }
}
```

Como puede verse, 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 del parámetro que recoge la *exception* que se produzca.

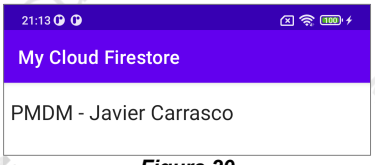


Figura 29

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.

```
docRef.get().apply {
    // Se obtiene la información una vez completada la conexión.
    addOnCompleteListener {
        if (it.isSuccessful) {
            // Documento encontrado en la caché offline.
            val doc = it.result

            Log.d("addOnCompleteListener", "Document data: ${doc?.data}")

            val texto = "${doc!!["modulo"]} - ${doc["nombre"]} ${doc["apellido"]}"
            binding.textView.text = texto
        } else Log.d("addOnCompleteListener", "Fallo de lectura ", it.exception)
    }
}
```

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()`.

```
// Escucha del documento, contrasta la caché con la base de datos.
docRef.addSnapshotListener { value, error ->
    if (error != null) {    // Se comprueba si hay fallo.
        Log.w("addSnapshotListener", "Escucha fallida!", error)
        return@addSnapshotListener
    }

    if (value != null && value.exists()) {
        Log.d("addSnapshotListener", "Información actual: ${value.data}")
        val texto = "${value["modulo"]} - ${value["nombre"]} ${value["apellido"]}"
        binding.textView.text = texto
    } else Log.d("addSnapshotListener", "Información actual: null")
}
```

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. En este método, el parámetro `value` contiene el documento leído.

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

```
// Obtener todos los documentos de una colección (sin escucha).
colProfesores.get().apply {
    addOnSuccessListener {
        for (doc in it) {
            Log.d("DOC", "${doc.id} => ${doc.data}")
            binding.textView.append(
                "${doc!!["modulo"]} - ${doc["nombre"]} ${doc["apellido"]}\n"
            )
        }
    }

    addOnFailureListener { exception ->
        Log.d("DOC", "Error durante la recogida de documentos: ", exception)
    }
}
```

El resultado que debe obtenerse será como el que se muestra en la figura. Pero, si se modifican datos en la base de datos, como este sistema no dispone de escucha activa, no podrán verse las modificaciones de manera inmediata.

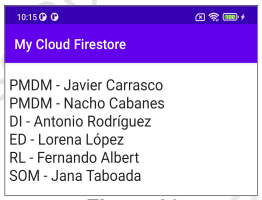


Figura 30

El siguiente método permite recuperar todos los documentos de una colección añadiendo escucha activa al proceso.

```
// Obtener todos los documentos de una colección (con escucha).
colProfesores.addSnapshotListener { querySnapshot, firestoreException ->
    if (firestoreException != null) {
        Log.w("addSnapshotListener", "Escucha fallida!.", firestoreException)
        return@addSnapshotListener
    }

    binding.textView.text = ""
    for (doc in querySnapshot!!) {
        Log.d("DOC", "${doc.id} => ${doc.data}")
        binding.textView.append("${doc["modulo"]} - ${doc["nombre"]} ${doc["apellido"]}\n")
    }
}
```

Ahora bien, seguramente no siempre se pretenda obtener todos los documentos de una colección. Para filtrar los resultados, se deberá interponer el método `where()` entre la colección y el sistema a utilizar para recuperar la información. Este método tiene tres parámetros, el campo por el que 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.

```
// ==, <, <=, >, >=
.whereEqualTo("state", "CA")
.whereLessThan("population", 100000)
.whereLessThanOrEqualTo("name", "San Francisco")
.whereGreaterThan("population", 100000)
.whereGreaterThanOrEqualTo("name", "San Francisco")
// Contenido en el Array.
.whereArrayContains("regions", "west_coast")
```

Si se quiere utilizar los métodos *where*, deberá utilizarse 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 como se ha comentado. A continuación, duplica el *TextView* del ejemplo como `textView2`, de forma que se puedan mostrar datos simultáneamente con y sin escucha, y añadiendo los métodos *where*.

```
// Obtiene todos los documentos de una colección filtrados (sin escucha).
colProfesores.whereEqualTo("modulo", "PMDM").get().apply {
    addOnSuccessListener {
        binding.textView.text = getString(R.string.txt_no_escucha)
        for (doc in it) {
            Log.d("DOC", "${doc.id} => ${doc.data}")
```

```
            binding.textView.append(
                "${doc["modulo"]} - ${doc["nombre"]} ${doc["apellido"]}\n"
            )
        }
    }

    addOnFailureListener { exception ->
        Log.d("DOC", "Error durante la recogida de documentos: ", exception)
    }
}

// Obtiene todos los documentos de una colección filtrados (con escucha).
colProfesores.whereEqualTo("modulo", "ED")
    .addSnapshotListener { querySnapshot, firebaseFirestoreException ->
        if (firebaseFirestoreException != null) {
            Log.w("addSnapshotListener", "Escucha fallida!.", firebaseFirestoreException)
            return@addSnapshotListener
        }

        binding.textView2.text = getString(R.string.txt_con_escucha)
        for (doc in querySnapshot!!) {
            Log.d("DOC", "${doc.id} => ${doc.data}")
            binding.textView2.append(
                "${doc["modulo"]} - ${doc["nombre"]} ${doc["apellido"]}\n"
            )
        }
    }
```

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

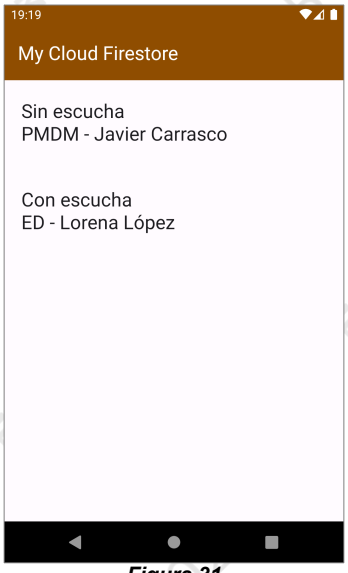


Figura 31

Si quieres, 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.

```
colProfesores.whereEqualTo("nombre", "Javier")
    .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 quieres 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.

```
binding.button.setOnClickListener {
    // Se crea la estructura del documento.
    val profe = hashMapOf(
        "nombre" to "Miguel",
        "apellido" to "López",
        "modulo" to "ED"
    )
    // Se añade el documento sin indicar ID, dejando que Firebase genere el ID
    // al añadir el documento. Para esta acción se recomienda add().
    colProfesores.document().set(profe)
        // Respuesta si ha sido correcto.
        .addOnSuccessListener {
            Log.d("DOC_SET", "Documento añadido!")
        }
        // Respuesta si se produce un fallo.
        .addOnFailureListener { e ->
            Log.w("DOC_SET", "Error en la escritura", e)
        }
}
```

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.

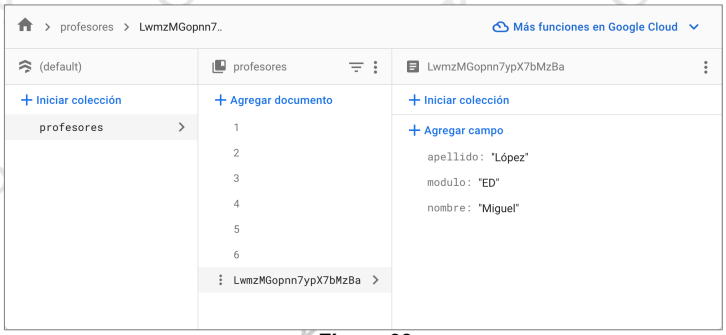


Figura 32

Si quieres especificar el *id*, deberás indicarlo, eso se hará en la instrucción `CollectionReference.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.

```
colProfesores.document("7").set(profe)
```

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:

```
colProfesores.document(contadorId.toString()).set(profe)
```

La variable `contadorId` se actualizará en el método `addSnapshotListener()` que se utiliza para completar el *TextView* `textView2`, 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()`.

```
val data = hashMapOf(
    "nombre" to "Javier",
    "apellido" to "Carrasco"
)
colProfesores.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"*. Se reutilizará el botón añadido para el nuevo código.

```
binding.button.setOnClickListener {
    val colModulos = db.collection("modulos")  // Crea la colección si no existe.
    val moduloNuevo = hashMapOf(
        "siglas" to "PMDM",
        "nombre" to "Programación Multimedia y Dispositivos Móviles"
    )

    colModulos.add(moduloNuevo)
        .addOnSuccessListener { Log.d("DOC_ADD", "Documento añadido, id: ${it.id}") }
        .addOnFailureListener { e -> Log.w("DOC_ADD", "Error añadiendo el documento", e) }
}
```

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

```
2022-... DOC_ADD  es.javiercarrasco.mycloudfirestore  D  Documento añadido, id: OKKpJwCJuFzzMSiL2LcC
```

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

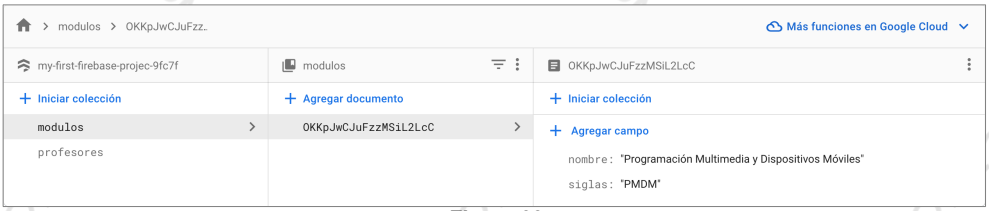


Figura 33

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.

```
binding.button.setOnClickListener {
    // Se crea un documento nuevo obteniendo su referencia.
    val refModuloNuevo = db.collection("modulos").document()

    val modulo = hashMapOf(
        "abreviatura" to "ED",
        "nombre" to "Entornos de Desarrollo"
    )
    refModuloNuevo.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 *OKKpJwCJuFzzMSiL2LcC*. Si se desea modificar el valor del campo `abreviatura`, deberá realizarse la siguiente operación.

```
binding.button.setOnClickListener {
   // Se obtiene la referencia del documento a actualizar.
   val refModuloUpd = db.collection("modulos").document("OKKpJwCJuFzzMSiL2LcC")

   refModuloUpd
      .update("abreviatura", "MÓVILES")
      .addOnSuccessListener { Log.d("DOC_UPD", "Documento actualizado correctamente") }
      .addOnFailureListener { e -> Log.w("DOC_UPD", "Error al actualizar el documento", e) }
}
```

Si es necesario actualizar varios campos al mismo tiempo, se utilizará un `mapOf()` en el método `update()`.

```
.update(
    mapOf(
        "abreviatura" to "MÓVILES",
        "nombre" to "Móviles con Kotlin"
    )
)
```

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.

```
binding.button.setOnClickListener {
    // Se obtiene la referencia del documento a eliminar.
    val refModuloDel = db.collection("modulos").document("KjJpFq1isfDUfzawZ3m4")

    refModuloDel
       .delete()
       .addOnSuccessListener { Log.d("DOC_DEL", "Documento eliminado correctamente") }
       .addOnFailureListener { e -> Log.w("DOC_DEL", "Error al eliminar el documento", e) }
}
```

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.

14.3. Tipos compuestos en Firebase

En Cloud Firestore se dispone de dos tipos de datos que pueden contener campos anidados, los **mapas** y los ***arrays***. En este punto se verá como crear este tipo de campos y como poder 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 más habitual:

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

Desde la consola de Firebase puedes crear una nueva colección y sus documentos como se muestra en la siguiente figura. Deberás seleccionar el tipo de datos *map* para crear el atributo “datos” y *array* para el atributo “provincias”. Para importar datos desde un fichero deberás disponer de un plan de pago. O como se verá a continuación, crearla desde el código de la aplicación.

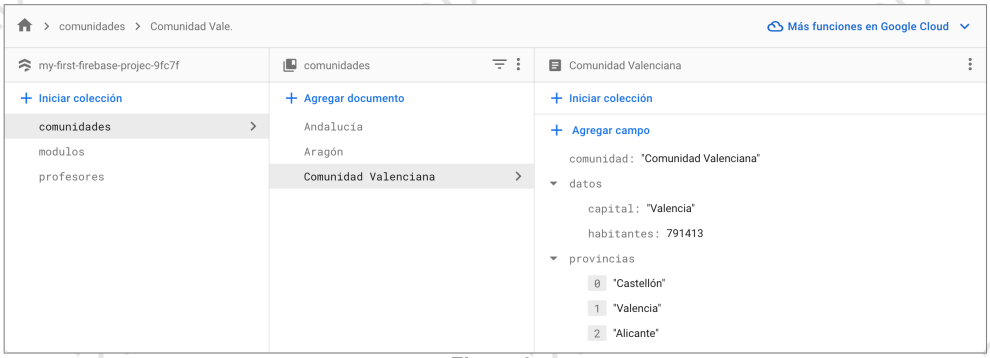


Figura 34

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. Se seguirá trabajando en el proyecto creado anteriormente, en la pestaña “General” de la “Configuración del proyecto” podrás registrar la nueva aplicación. Recuerda añadir las dependencias para la configuración del proyecto en Android Studio como en ejemplos anteriores y el permiso de acceso a Internet.

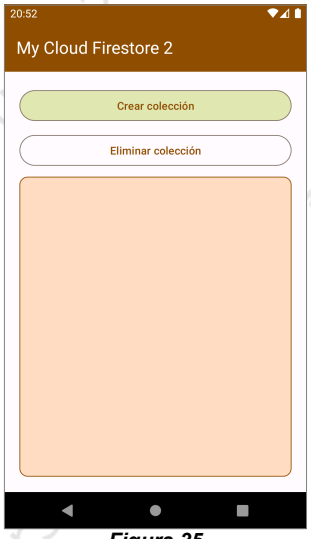


Figura 35

En este ejemplo se creará una nueva aplicación, *My Cloud Firestore 2*. Esta se limitará a crear documentos con los tipo de datos presentados en este punto, mostrarlos y eliminarlos. En la figura que aparece junto al texto puedes ver el aspecto que podría tener.

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 se detalla a continuación.

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

```
1  implementation("com.google.firebase:firebase-firestore:24.8.1")
```

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

```
1  implementation("com.google.firebase:firebase-firestore-ktx:24.8.1")
```

Puedes utilizar cualquiera de las dos librerías, pero para este ejemplo se utilizará la segunda opción. Volviendo al código, en primer lugar se crearán las siguiente dos propiedades para la clase `MainActivity`.

```
private val db: FirebaseFirestore = FirebaseFirestore.getInstance()
private val collComunidades = db.collection("comunidades")
```

También se creará el siguiente método para eliminar el documento indicado por el *id*. Recuerda que no se puede, o debe, eliminar colecciones completas, por lo que se eliminará documento a documento.

```
// Borra el documento indicando el ID por parámetro.
fun borrarDocumento(id: String) {
    // Elimina documento a documento, para ello, se necesita conocer su identificador.
    colComunidades.document(id)
        .delete()
        .addOnSuccessListener { Log.d("DOC_DEL", "Documento eliminado correctamente") }
        .addOnFailureListener { e -> Log.w("DOC_DEL", "Error al eliminar el documento", e) }
}
```

En el método `onCreate()` se añadirá el código para recuperar la información de la colección.

```
// Se obtienen todos los documentos de la colección (con escucha) y se rellena el TextView.
colComunidades.addSnapshotListener { querySnapshot, e ->
    if (e != null) {
        Log.w("DOC_SHOW", "Escucha fallida!.", e)
        return@addSnapshotListener
    }

    binding.textView.text = ""
    for (document in querySnapshot!!) {
        Log.d("DOC_SHOW", "${document.id} => ${document.data}")
        binding.textView.append(
            "${document!!["comunidad"]}\n" +
                    "\tCapital: ${document["datos.capital"]}\n" +
                    "\tHabs: ${document["datos.habitantes"]}\n" +
                    "\tProvincias: ${document["provincias"]}\n\n"
        )
    }
}
```

A continuación, en el método `onStart()`, se crearán los *listeners* para los botones, empezando por el botón **Crear colección**.

```
// Botón para añadir documentos a la colección.
binding.btnCreate.setOnClickListener {
    // Se prepara la estructura de datos.
    val comunidades = listOf(
        mapOf( // Comunidad 1
            "comunidad" to "Andalucía",
            "datos" to mapOf(
                "capital" to "Sevilla",
                "habitantes" to 688711
            ),
            "provincias" to listOf(
```

```
                "Almería", "Granada", "Córdoba", "Jaén",
                "Sevilla", "Málaga", "Cádiz", "Huelva"
            )
        ), mapOf( // Comunidad 2
            "comunidad" to "Comunidad Valenciana",
            "datos" to mapOf(
                "capital" to "Valencia",
                "habitantes" to 791413
            ),
            "provincias" to listOf("Castellón", "Valencia", "Alicante")
        ), mapOf( // Comunidad 3
            "comunidad" to "Aragón",
            "datos" to mapOf(
                "capital" to "Zaragoza",
                "habitantes" to 666880
            ),
            "provincias" to listOf("Huesca", "Zaragoza", "Teruel")
        )
    )

    // Se añade documento a documento.
    for (doc in comunidades) {
        Log.d("DOC_ADD", "Añadiendo documento " + doc["comunidad"])
        colComunidades.document(doc["comunidad"].toString()).set(doc)
            .addOnSuccessListener { Log.d("DOC_ADD", "Documento añadido correctamente") }
            .addOnFailureListener { e ->
                Log.w(
                    "DOC_ADD",
                    "Error al añadir el documento", e
                )
            }
    }
}
```

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

```
// Botón para eliminar documentos de la colección.
binding.btnDelete.setOnClickListener {
    borrarDocumento("Andalucía")
    borrarDocumento("Aragón")
    borrarDocumento("Comunidad Valenciana")
}
```

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

```
// Se obtiene la referencia al documento
val refComunidad = colComunidades.document("Aragón")
// Se añade un valor nuevo al array provincias.
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*, en este caso se utilizaría el método `arrayRemove()`.

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

Ahora fíjate como podrían obtenerse los datos completos de un *array* y tratarlos como tal.

```
colComunidades.document("Aragón").get().apply {
    addOnSuccessListener {
        if (it.exists()) {
            val datos = it.get("provincias") as ArrayList<*>
            binding.textView.append("${datos[2]}")
        }
    }
}
```

Por último, observa como se obtendrían los datos del mapa que contiene la información de la capital, de tal manera que pudiese tratarse como un *HashMap*.

```
colComunidades.document("Aragón").get().apply {
    addOnSuccessListener {
        if (it.exists()) {
            val datos = it["datos"] as HashMap<*, *>
            binding.textView.append("${datos["capital"]}: ${datos["habitantes"]}")
        }
    }
}
```

Presta atención en la condición del *if*, deberás comprobar si existe el documento para poder utilizarlo, en caso contrario, la aplicación fallará. Esta operación también puede hacerse utilizando la escucha activa.

14.4. Autenticación con Firebase

De sobra es conocido que cada vez son más las aplicaciones que necesitan almacenar información de los usuarios en la nube. El objetivo básico, suele ser proporcionar la misma experiencia de uso en diferentes dispositivos.

Firebase Authentication ofrece una serie de herramientas para *backend*, un SDK y bibliotecas de IU para facilitar su aplicación. Entre los tipos de autenticación disponibles se puede encontrar la autenticación mediante contraseña, número de teléfono o identidad federada (Google, Facebook, Twitter, etc).

Esta utilidad de Firebase está totalmente integrada con otros servicios de la plataforma e implementa los principales estándares de autenticación como OAuth 2.0 y OpenID Connect.

Se dispone dos forma de hacer uso de **Firebase Authentication**, la primera, **FirebaseUI Auth**, esta opción es la recomendada, agregando el sistema completo de autenticación a la aplicación, proporcionando una autenticación directa, permitiendo el control de los flujos de IU, autenticación mediante correo electrónico y contraseña, números de teléfonos e identidad federada. La segunda opción, **mediante SDK**, permite implementar únicamente la parte, o partes, que se necesiten, autenticación mediante correo electrónico y contraseña, autenticación mediante identidad federada, por número de teléfono o de forma anónima.

El funcionamiento de Firebase Authentication será el siguiente, cuando un usuario quiere acceder a la aplicación, deberá obtenerse en primer lugar sus credenciales, ya bien sean mediante correo electrónico y contraseña, número de teléfono o identificación federada (***token OAuth***). Una vez obtenidas las credenciales, deberán pasarse a Firebase Authentication, el *backend* verificará las credenciales y devolverá una respuesta.

Es importante saber que el acceso por defecto mediante Firebase Authentication, permite a los usuarios autenticados leer y escribir datos en Firebase Realtime Database y Cloud Storage, por lo que deberán modificarse las reglas de acceso si no es los que interesa.

14.4.1. Configuración del proyecto

Para probar la autenticación de Firebase se creará un nuevo proyecto (*My Auth Firebase*) añadiéndolo al proyecto Firebase creado en este tema. Añade las dependencias de Firebase y el fichero `google-services.json` al proyecto nuevo. Recuerda añadir el permiso de internet en el *manifest*.

Una vez añadidas las dependencias básicas de Firebase, se añadirá la librería necesaria para hacer uso de Firebase Authentication.

```
1  implementation("com.google.firebase:firebase-auth-ktx:22.1.2")
```

Para poder hacer uso de Firebase Authentication deberá habilitarse desde la consola Firebase del proyecto, desde la sección **Compilación** seleccionando la opción **Authentication**.

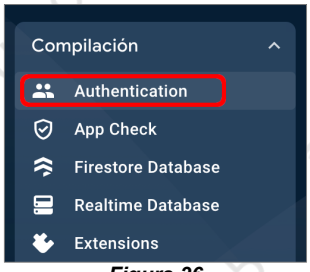


Figura 36

Tras activar la opción, ya puedes comenzar con la configuración de la autenticación.

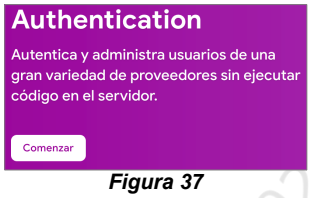


Figura 37

A continuación, en la opción ***Sign-in method***, que aparecerá seleccionada, deberás elegir el primer método de autenticación.

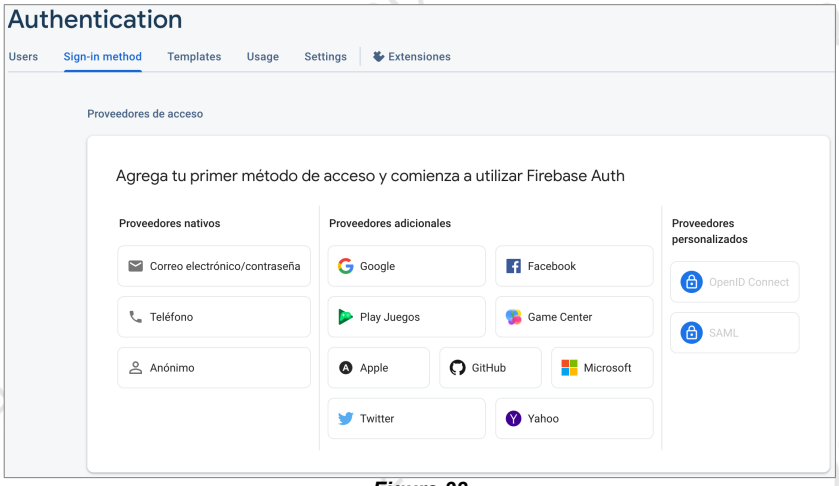


Figura 38

Se utilizará para este ejemplo un proveedor nativo, **Correo electrónico/contraseña**. Selecciona la opción para acceder a la configuración.

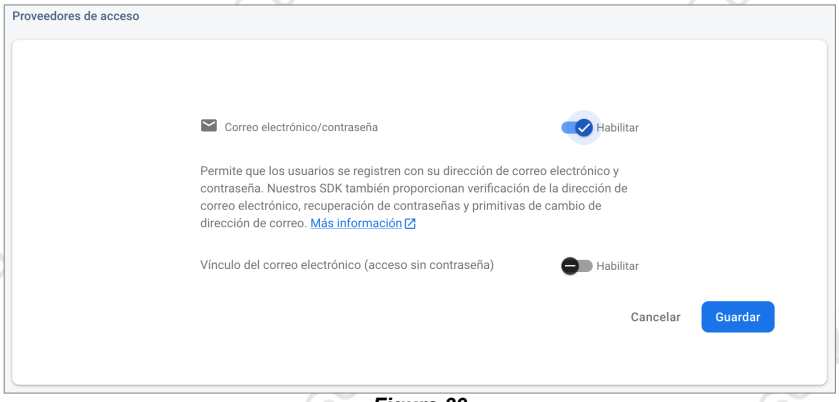


Figura 39

Habilita la opción Correo electrónico/contraseña y pulsa el botón guardar. Tras un breve proceso, volverás a la pantalla anterior y podrás ver que el proveedor de acceso está activado.

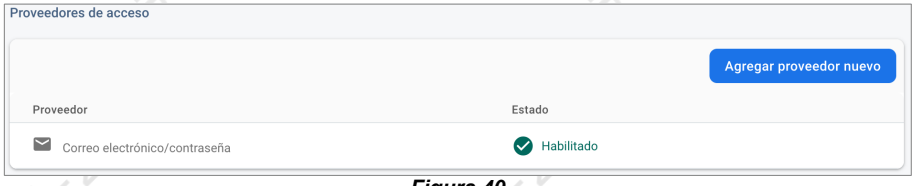


Figura 40

Una vez habilitado, en la pestaña **Users**, podrás ver un listado con todos los usuarios que vayan autenticándose en la aplicación.

Figura 41

Ahora ya estaría el servicio activado, por lo que se volverá al proyecto de Android Studio para continuar.

14.4.2. Comprobar el estado de la autenticación

El primer paso será crear el objeto encargado de la autenticación en la clase principal del proyecto.

```
private lateinit var auth: FirebaseAuth
```

En el método *onCreate* se realizará la instanciación.

```
auth = FirebaseAuth.getInstance()
```

Esta vez, en el método *onStart* se hace la comprobación del registro o *login* del usuario.

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

    // Si el usuario existe NO será nulo.
    val currentUser = auth.currentUser
    updateUI(currentUser)
}
```

El método **updateUI** se encargará de comprobar si *currentUser* es nulo, en tal caso, el usuario no está registrado. Puedes omitir la creación de la variable *currentUser* si lo crees conveniente.

```
// Método para comprobar el estado del usuario actual,
// si está registrado su valor no será nulo.
private fun updateUI(user: FirebaseUser?) {
    if (user != null)
        binding.tvInfo.text = getString(R.string.txt_user_login)
    else binding.tvInfo.text = getString(R.string.txt_user_no_login)
}
```

14.4.3. Registrar un usuario

A continuación, se añadirán a la UI dos botones, uno que permita el inicio de sesión y otro para realizar el registro.

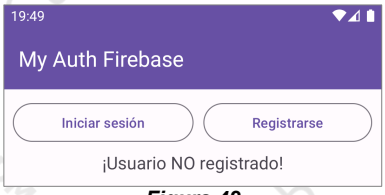


Figura 42

Como no hay usuarios creados, no se podrá iniciar sesión. En primer lugar se habilitará el registro de usuarios. Al pulsar cualquiera de los dos botones se lanzará un cuadro de diálogo para solicitar el correo electrónico y una contraseña (código al final del capítulo), para ilustrar este ejemplo se omitirán las típicas comprobaciones como, introducir dos veces la contraseña o el correo electrónico. Para el registro, se creará el método ***createAccount*** para tal efecto.

```
// Método encargado de crear la cuenta de usuario mediante email y password.
private fun createAccount(email: String, password: String) {
    Log.d(TAG, "createAccount:$email")

    auth.createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener(this) { task ->
            if (task.isSuccessful) {
                // Alta correcta, se actualiza la UI con la nueva información.
                Log.d(TAG, "createUserWithEmail:success")
                updateUI(auth.currentUser)
            } else {
                // Fallo, se muestra mensaje al usuario.
                Log.w(TAG, "createUserWithEmail:failure", task.exception)
                Toast.makeText(
                    baseContext, "Authentication failed",
                    Toast.LENGTH_SHORT
                ).show()
                updateUI(null)
            }
        }
}
```

Si te fijas detenidamente en este método, la creación de un usuario nuevo depende del método ***createUserWithEmailAndPassword*** del objeto *auth*, y recuerda que éste es una instancia de la clase *FirebaseAuth*. A continuación, se evalúa la respuesta recibida a la tarea mediante un *listener*, tras lo cual se actualizará la interfaz de usuario. Se añadirá el *listener* al botón correspondiente en *onStart* y listo.

```
private fun updateUI(user: FirebaseUser?) {
    if (user != null) {
        binding.tvInfo.text = getString(R.string.txt_user_login)
```

```
        binding.btnLogin.text = getString(R.string.txt_btn_logout)
        binding.btnRegister.text = getString(R.string.txt_btn_verify_email)
    } else {
        binding.tvInfo.text = getString(R.string.txt_user_no_login)
        binding.btnLogin.text = getString(R.string.txt_btn_login)
        binding.btnRegister.text = getString(R.string.txt_btn_register)
    }
}
```

Como puedes ver en el método *updateUI*, se actualiza el texto de los botones, esto se hace para comprobar en que estado se encuentra el botón para lanzar una tarea u otra.

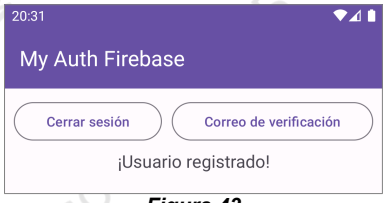


Figura 43

14.4.4. Cerrar sesión

Por ejemplo, el *listener* para el botón de inicio de sesión se podría dejar de la siguiente forma, dónde también puedes ver como se haría el **cierre de sesión** de un usuario autenticado.

```
binding.btnLogin.setOnClickListener {
    if (auth.currentUser != null) {
        auth.signOut()
        updateUI(null)
    }else showLoginDialog(R.string.txt_btn_login)
}
```

14.4.5. Verificación del correo

Para la verificación del correo electrónico se creará el método ***sendEmailVerification***.

```
// Método encargado de enviar el email de verificación al usuario.
private fun sendEmailVerification() {
    // Deshabilita el botón de verificación.
    binding.btnRegister.isEnabled = false

    // Envío email verificación
    val user = auth.currentUser
    user?.sendEmailVerification()
        ?.addOnCompleteListener(this) { task ->
            // Se habilita el botón
            binding.btnRegister.isEnabled = true

            if (task.isSuccessful) {
                Toast.makeText(
```

```
                    baseContext,
                    "Email de verificación enviado a ${user.email} ",
                    Toast.LENGTH_SHORT
                ).show()
            } else {
                Log.e(TAG, "sendEmailVerification", task.exception)
                Toast.makeText(
                    baseContext,
                    "Fallo en el envío del email de verifiación.",
                    Toast.LENGTH_SHORT
                ).show()
            }
        }
}
```

Al igual que ocurre al crear una nueva cuenta de usuario, el método que realmente hace la operación es ***sendEmailVerification***, y únicamente deberá tratarse con el valor devuelto, en este caso, si se ha enviado o no correctamente la notificación.

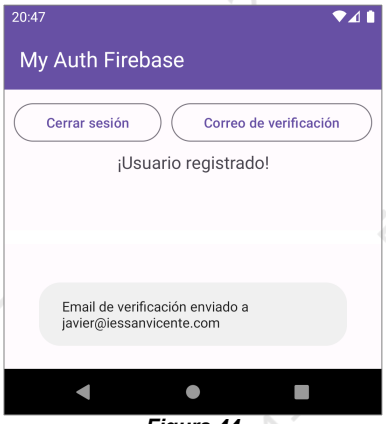


Figura 44

Si el envío del correo de verificación se ha realizado correctamente, lo hace Firebase, deberá recibirse en el buzón un correo de verificación.

```
Hello,

Follow this link to verify your email address.

https://my-first-firebase-projec-8738d.firebaseapp
.com/__/auth/action?mode=verifyEmail&oobCode=gt3I552EkrA9Xn63
tqrHrt6XaT4lBxqqe35xM9bQXVEAAAGLNKoG0Q&apiKey=AIzaSyBMPY1rnrPHB9o
lAhcyeX4ogo9lpQrJ1mg&lang=en

If you didn't ask to verify this address, you can ignore this email.

Thanks,

Your my-first-firebase-projec-8738d team
```

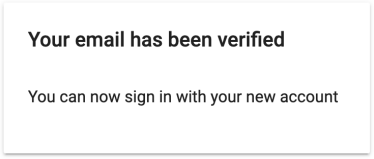


Figura 45

Ésta es la plantilla por defecto, pero si quieres personalizarla[7], deberás hacerlo desde la consola de Firebase, dentro de *Authentication*, en la sección Plantillas (*Templates*).

Tras seguir el enlace, podrás ver una página en el navegador, indicando que se ha producido la verificación correctamente.

Una vez verificado el correo de la cuenta, se puede modificar el método *updateUI* para que se adapte a la nueva información.

7 Personalizar plantillas (https://support.google.com/firebase/answer/7000714)

```
private fun updateUI(user: FirebaseUser?) {
    if (user != null) {
        binding.tvInfo.text = getString(R.string.txt_user_login)
        binding.tvInfo.append(
            getString(
                R.string.txt_info_user,
                user.email,
                user.isEmailVerified,
                user.uid
            )
        )
        binding.btnLogin.text = getString(R.string.txt_btn_logout)

        // Se habilita el botón de verificación si el email está no verificado.
        binding.btnRegister.isEnabled = !user.isEmailVerified
        binding.btnRegister.text = getString(R.string.txt_btn_verify_email)

    } else {
        binding.tvInfo.text = getString(R.string.txt_user_no_login)
        binding.btnLogin.text = getString(R.string.txt_btn_login)
        binding.btnRegister.text = getString(R.string.txt_btn_register)
        binding.btnRegister.isEnabled = true
    }
}
```

Como información, la variable `R.string.txt_info_user` está formada de la siguiente forma.

```
<string name="txt_info_user">
    Email: %1$s\nCorreo verificado: %2$b\nUsuario Firebase: %3$s
</string>
```

Fíjate que se suele utilizar %s para indicar que el único argumento es un *string*, pero en esta variable se ha utilizado **%1$s** para indicar que el primer argumento es *string*, y **%2$b** para indicar que el segundo argumento es un boolean.

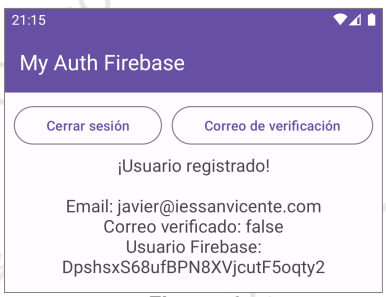


Figura 46

Llegados a este punto, en la consola de Firebase en la sección de ***Authentication***, se podrá ver el listado de todos los usuarios que vayan registrándose en la aplicación.

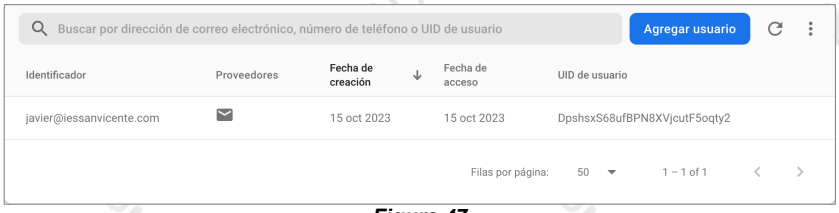


Figura 47

14.4.6. Iniciar sesión

Por último, se habilitará el inicio de sesión de un usuario ya registrado, y como en los pasos anteriores, Firebase facilita enormemente esta tarea. El método en cuestión será como sigue:

```
// Método para iniciar sesión con una cuenta ya existente.
private fun signIn(email: String, password: String) {
    Log.d(TAG, "signIn:$email")

    auth.signInWithEmailAndPassword(email, password)
        .addOnCompleteListener(this) { task ->
            if (task.isSuccessful) {
                // Login, se actualiza la UI con la información del usuario.
                Log.d(TAG, "signInWithEmail:success")
                updateUI(auth.currentUser)
            } else {
                // Login fallido, se muestra un mensaje de error.
                Log.w(TAG, "signInWithEmail:failure", task.exception)
                Toast.makeText(
                    applicationContext,
                    "Authentication failed ${task.exception!!.message}",
                    Toast.LENGTH_SHORT
                ).show()
                updateUI(null)
            }
        }
}
```

Al igual que ocurre con los métodos vistos anteriormente, el método que realmente hace la operación es ***signInWithEmailAndPassword***, el cual lleva como parámetros el email y la contraseña introducida por el usuario, únicamente deberá tratarse con el valor devuelto, en este caso será, si la sesión se ha iniciado o no correctamente.

Para terminar esta parte se mostrará como queda el método ***showLoginDialog***, encargado de mostrar el cuadro de diálogo según el botón pulsado.

```
private fun showLoginDialog(title: Int) {
    val bindCustomDialog = LoginLayoutBinding.inflate(layoutInflater)

    val dialogLogin = MaterialAlertDialogBuilder(this).apply {
        setView(bindCustomDialog.root)
```

```
        setTitle(title)
        setPositiveButton(android.R.string.ok, null)
        setNegativeButton(android.R.string.cancel) { dialog, _ ->
            dialog.cancel()
        }
    }.create()

    dialogLogin.setOnShowListener {
        dialogLogin.getButton(AlertDialog.BUTTON_POSITIVE).setOnClickListener {
            val email = bindCustomDialog.etEmail.text.toString().trim()
            val pass = bindCustomDialog.etPassword.text.toString().trim()

            if (email.isNotEmpty() && pass.isNotEmpty()) {
                if (title == R.string.txt_btn_register)
                    createAccount(email, pass)
                else signIn(email, pass)

                dialogLogin.dismiss()
            } else {
                bindCustomDialog.etEmail.error = getString(R.string.txtErrorDialog)
                bindCustomDialog.etPassword.error = getString(R.string.txtErrorDialog)
            }
        }
    }
    dialogLogin.show()
}
```

14.4.7. Uso del Token

En este ejemplo se ha utilizado el **UID** del usuario, fíjate que se muestra cuando está la sesión iniciada. Pero, este UID es único sólo para el proyecto, si se desea que los cliente se verifiquen contra un servidor *backend* personalizado, deberá recuperarse su ***token*** una vez iniciada sesión. Se pueden añadir las siguientes líneas en el método ***updateUI*** para recuperar el *token* del usuario.

```
if (user != null) {
    ...

    user.getIdToken(true)
        .addOnCompleteListener { task ->
            if (task.isSuccessful) {
                val idToken = task.result?.token
                binding.tvInfo.append(getString(R.string.txt_token_user, idToken))
            } else {
                binding.tvInfo.append(getString(R.string.txt_token_fail))
            }
        }
```

El resultado los puedes ver en la siguiente figura, recuerda guardar el *token* para mantener la sesión.

Puedes encontrar más información en el repositorio GitHub de inicio rápido de Firebase[8]. Concretamente en la sección *Auth*, dónde podrás encontrar varios ejemplos de autenticación.

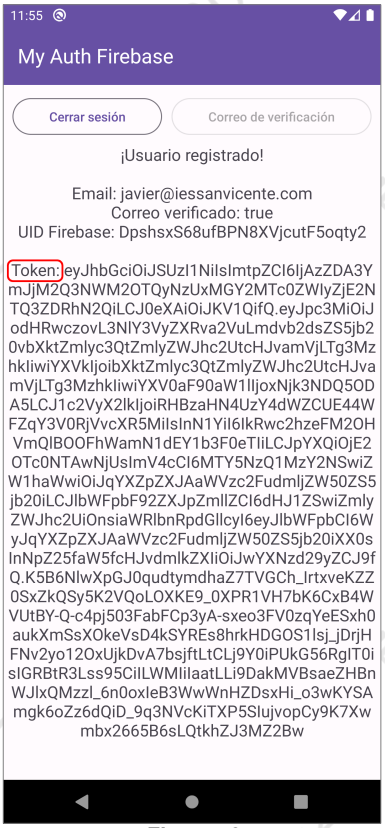


Figura 48

14.5. Cloud Storage

Cloud Storage es un servicio que ofrece Google para el almacenamiento de archivos u otro tipo de objetos en la nube, permitiendo a las aplicaciones, mediante el uso de su SDK, la carga y descarga a través de la red.

14.5.1. Configuración del proyecto

Para probar Cloud Storage se creará un nuevo proyecto (*My Cloud Storage*) añadiéndolo al proyecto Firebase creado en este tema. Añade las dependencias de Firebase y el fichero `google-services.json` al proyecto nuevo. Recuerda añadir el permiso de uso de internet en el *manifest*.

Una vez añadidas todas las dependencias básicas de Firebase, se añadirá la librería necesaria para poder utilizar el servicio Cloud Storage.

Esta vez, para ver otro sistema, se indicará la dependencia sin indicar la versión, de esta forma se hará uso de **BoM de Firebase para Android**, así la aplicación siempre utilizará versiones compatibles de las bibliotecas de Firebase.

```
1  implementation("com.google.firebase:firebase-storage-ktx")
```

Para poder hacer uso de Cloud Storage sin necesidad de autenticarse deberá activarse en Firebase Authentication desde la consola Firebase del proyecto (**Authentication**) la posibilidad de acceso anónimo.

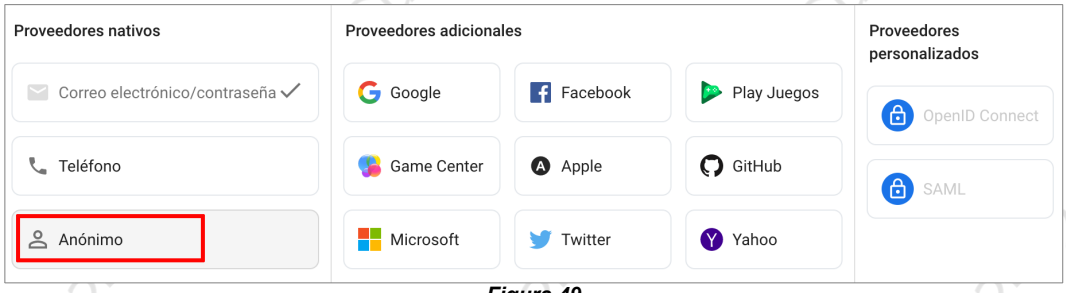


Figura 49

8 Firebase Quickstarts for Android (https://github.com/firebase/quickstart-android)

Una vez activado, ya estás en disposición de activar **Storage** desde la sección **Compilación**.

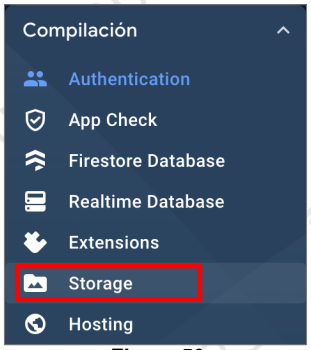


Figura 50

Figura 51

Tras pulsar el botón “Comenzar” se lanzará un asistente en que se deberá elegir el tipo de acceso, se dejará marcada la opción **“modo producción”**, observa la regla presentada, habrá que cambiar el valor *false* por *true*. Y la ubicación, que se quedará en este caso en **eur3**, elección hecha en la creación del proyecto. Una vez terminado podrás ver la consola de Storage.

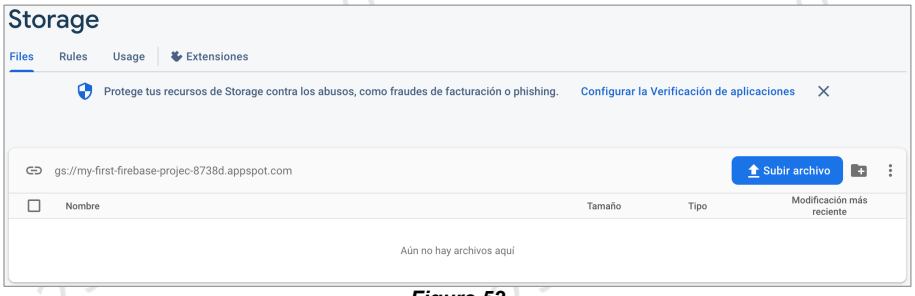


Figura 52

14.5.2. Comprobar el estado de Storage

En el proyecto de Android Studio se hará una comprobación del estado del servicio, muy parecido al hecho para la autenticación. Se creará la siguiente propiedad.

```
1  private lateinit var storage: FirebaseStorage
```

Y en el método *onCreate* se obtendrá la instancia al servicio.

```
1  storage = FirebaseStorage.getInstance()
```

Con estas líneas ya puedes probar y ver en el *logcat* si se estable la conexión con Firebase.

14.5.3. Crear referencias

El siguiente paso consistirá en crear referencias[9], estas se utilizarán para ubicar los ficheros subidos a Firebase, y para navegar por la estructura creada. De esta forma puedes crear la estructura de directorios que necesites.

9 Crear referencias (https://firebase.google.com/docs/storage/android/create-reference)

```
val storageRef: StorageReference = storage.reference // Raíz del Storage
val imagesRef: StorageReference = storageRef.child("images") // Carpeta images
val fileImageRef: StorageReference = imagesRef.child("prueba.jpg") // Fichero prueba.jpg
```

Si muestras la información de estas variables en el *log*...

```
Log.i("storageRef", "$storageRef")
Log.i("imagesRef", "$imagesRef")
Log.i("fileImageRef", "$fileImageRef")
```

... puedes obtener algo similar a esto.

```
2023...7 storageRef   ...o.mycloudstorage  I  gs://my-...appspot.com/
2023...7 imagesRef    ...o.mycloudstorage  I  gs://my-...appspot.com/images
2023...7 fileImageRef ...o.mycloudstorage  I  gs://my-...appspot.com/images/prueba.jpg
```

Además, puedes a partir de un punto, por ejemplo, con la variable `fileImageRef`, hacer uso de `parent`.

```
fileImageRef.parent
```

Con esto se obtendría la carpeta que contiene el fichero en Firebase. Pero, todavía no estaría cargado el fichero a Firebase.

14.5.4. Subir un fichero

El siguiente paso consistirá en subir un fichero a Firebase[10], para este ejemplo, se subirá la imagen cargada en un *ImageView* a Firebase.

```
val bitmap = (binding.imageView.drawable as BitmapDrawable).bitmap
val baos = ByteArrayOutputStream()
bitmap.compress(Bitmap.CompressFormat.JPEG, 100, baos)
val data = baos.toByteArray()

val uploadTask: UploadTask = fileImageRef.putBytes(data)
uploadTask.addOnFailureListener {
    Log.d("addOnFailureListener", "Fallo en la carga ", it)
}.addOnSuccessListener {
    Log.d("addOnSuccessListener", "Uploaded file: ${it.metadata?.sizeBytes}")
}
```

El objeto ***uploadTask*** será el encargado de realizar la tarea, además de añadir dos *listener*, uno para tener controlado un posible error, y otro que se ejecutará cuando la tarea se ha completado correctamente.

```
2023...7 addOnSuccessListener    es.javiercarrasco.mycloudstorage     D  Uploaded file: 7684
```

Puedes ampliar información sobre diferentes tipos de subida de archivos a Firebase en su documentación.

10 Subir un fichero (https://firebase.google.com/docs/storage/android/upload-files)

14.5.5. Descargar archivos

Otra de las acciones más habituales si se utiliza **Cloud Storage** es la descarga de archivos[11]. Cuando un fichero es descargado, se crea un fichero temporal, a partir de ese punto, se tratará como se desee. En el siguiente ejemplo, se muestra como se descarga una imagen y, se almacena en la carpeta *files* de la aplicación (almacenamiento local), mostrando la imagen en un *ImageView* tras la descarga.

Al igual que para subir un archivo, será necesario obtener la referencia del archivo que se desea descargar.

```
1  val fileArbolRef: StorageReference = imagesRef.child("arbol.jpg") // Fichero arbol.jpg
```

Se prepara el fichero temporal para almacenar la descarga.

```
2  val localFile: File = File.createTempFile("images", "jpg")
```

A continuación, se añaden los *listener* para la descarga, tanto para la descarga completa como para prevenir un posible fallo.

```
3  fileArbolRef.getFile(localFile).addOnSuccessListener {
4
5      val salida: OutputStream
6      try { // Se guarda el fichero temporal en la carpeta files
7          salida = openFileOutput(
8              fileArbolRef.name,
9              Activity.MODE_PRIVATE
10         )
11         salida.write(localFile.readBytes())
12         salida.flush()
13         salida.close()
14
15         localFile.delete() // Borra el fichero temporal
16     } catch (e: Exception) {
17         Log.d("Exception", "Error al escribir el fichero", e)
18     }
19
20     Glide.with(this)
21         .load("${filesDir.path}/${fileArbolRef.name}") // Carga la imagen desde files
22         .override(binding.imageView.width)
23         .fitCenter()
24         .into(binding.imageView)
25
26 }.addOnFailureListener {
27     Log.d("addOnFailureListener", "Fallo en la descarga ", it)
28 }
```

11 Descargar archivos (https://firebase.google.com/docs/storage/android/download-files)