

Apuntes

[Descargar estos apuntes](#)

Tema 11. Persistencia de Datos I

Índice

1. [Acceso a bases de datos locales. SQLite](#)
 1. [Introducción](#)
 2. [Creación y apertura de la BD](#)
 3. [Acceso y modificación de los datos en la BD](#)
 1. [Acceso a la BD mediante sentencias sql](#)
 2. [Acceso a la BD mediante SQLiteDatabase](#)
 3. [Recuperación de datos con.rawQuery](#)
 4. [Recuperación de datos con query](#)
 4. [Uso de SimpleCursorAdapter](#)
 1. [Cursores con RecyclerView](#)
2. [Acceso a bases de datos remotas](#)
 1. [Introducción](#)
 2. [Acceso a bases de datos con Apirest y Retrofit](#)
 1. [Definición del Servicio Rest](#)
 2. [Consumo de un Servicio Rest desde Android](#)
3. [Acceso a Base de Datos con FIREBASE](#)
 1. [Creando un app de Firebase](#)
 2. [Firestore DataBase](#)
 1. [FirebaseUI y RecyclerView](#)
 2. [Filtrado y ordenación](#)
 3. [Firebase Auth. Autenticación](#)

Acceso a bases de datos locales. SQLite

Introducción

La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados: la base de datos **SQLite** y **Content Providers** (lo trabajaremos en temas posteriores).

Vamos a centrarnos en **SQLite**, aunque no entraremos ni en el diseño de BBDD relacionales ni en el uso avanzado de la BBDD. Para conocer toda la funcionalidad de SQLite se recomienda usar la documentación oficial.

Para el acceso a las bases de datos tendremos tres clases que tenemos que conocer. La clase **SQLiteOpenHelper**, que encapsula todas las funciones de creación de la base de datos y versiones de la misma. La clase **SQLiteDatabase**, que incorpora la gestión de tablas y datos. Y por último la clase **Cursor**, que usaremos para movernos por el recordset que nos devuelve una consulta *SELECT*.

Creación y apertura de la BD

El método recomendado para la creación de la base de datos es extender la clase **SQLiteOpenHelper** y sobrescribir los métodos **onCreate** y **onUpgrade**. El primer método se utiliza para crear la base de datos por primera vez y el segundo para actualizaciones de la misma. Si la base de datos ya existe y su versión actual coincide con la solicitada, se realizará la conexión a ella.

Para ejecutar la sentencia SQL utilizaremos el método **execSQL** proporcionado por la API para SQLite de Android.

```
internal inner class BDClientes(  
    context: Context?,  
    name: String?,  
    factory: CursorFactory?,  
    version: Int) : SQLiteOpenHelper(context, name, factory, version)  
{  
    var sentencia =  
        "create table if not exists clientes" +  
        "(dni VARCHAR PRIMARY KEY NOT NULL, nombre TEXT, apellidos TEXT);"  
    override fun onCreate(sqliteDatabase: SQLiteDatabase) {  
        sqliteDatabase.execSQL(sentencia)  
    }  
    override fun onUpgrade(sqliteDatabase: SQLiteDatabase, i: Int, i2: Int) {}  
}
```

✦ Si la base de datos existe pero su versión actual es anterior a la solicitada, se llamará automáticamente al método `onUpgrade()`. Si la base de datos no existe se llama automáticamente a `onCreate()`.

Acceso y modificación de los datos en la BD

Una vez creada la clase, instanciaremos un objeto del tipo creado y usaremos uno de los dos siguientes métodos para tener acceso a la gestión de datos:

`getWritableDatabase()` para acceso de escritura y `getReadableDatabase()` para acceso de solo lectura. En ambos casos se devolverá un objeto de la clase `SQLiteDatabase` para el acceso a los datos.

Para realizar una codificación limpia y reutilizable, lo mejor es crear una clase con todos los métodos que necesitemos para trabajar con la Base de Datos, además es útil tener la clase derivada de `SQLiteOpenHelper` interna a esta.

```
class BDAdapter(context: Context?)
{
    private var clientes: BDClientes

    init {
        clientes = BDClientes(context, "BDClientes", null, 1)
    }

    ...

    internal inner class BDClientes(
        context: Context?,
        name: String?,
        factory: CursorFactory?,
        version: Int) : SQLiteOpenHelper(context, name, factory, version)
    {
        var sentencia =
            "create table if not exists clientes" +
            "(dni VARCHAR PRIMARY KEY NOT NULL, nombre TEXT, apellidos TEXT);"
        override fun onCreate(sqliteDatabase: SQLiteDatabase) {
            sqliteDatabase.execSQL(sentencia)
        }
        override fun onUpgrade(sqliteDatabase: SQLiteDatabase, i: Int, i2: Int)
    }
}
```

✦ Como podemos ver en el código, creamos un objeto del tipo de `BDClientes` con el que vamos a trabajar llamando al constructor `BDClientes()`, a este método le pasamos el contexto, el nombre de la BBDD, un objeto cursor (con valor null) y la versión de la BD que necesitamos.

Para poder usar los métodos creados en la clase, tan solo instanciaremos un objeto de esta pasándole el contexto (de tipo activity). Con esta instancia podremos llamar a cualquiera de los métodos que crearemos en la clase de utilidad.

```
var bdAdapter = BDAdapter(this)
bdAdapter.insertarDatos()
```

Acceso a la BD mediante sentencias sql

El acceso a la BBDD se hace en dos fases, primero se consigue un objeto del tipo `SQLiteDatabase` y posteriormente usamos sus funciones para gestionar los datos. Si hemos abierto correctamente la base de datos entonces podremos empezar con las inserciones, actualizaciones, borrado, etc. No deberemos olvidar cerrar el acceso a la BD, siempre y cuando no se esté usando un cursor.

*Por ejemplo, vamos a insertar 10 registros en nuestra tabla clientes. Para ello podremos crear un método en el **BDAdapter** parecido al siguiente:*

```
fun insertarDatos() {
    var dbClientes = clientes.writableDatabase
    if (dbClientes != null) {
        for (i in 0..9) {
            val sentencia = "INSERT INTO Clientes (dni, nombre, apellidos) V
                " ('" + i + "', 'nombre" + i + "', 'apellido" + i + "');"
            dbClientes.execSQL(sentencia)
        }
        dbClientes.close()
    }
}
```



Vale, ¿y ahora qué? ¿Dónde está la base de datos que acabamos de crear? ¿Cómo podemos comprobar que todo ha ido bien y que los registros se han insertado correctamente?, en primer lugar veamos dónde se ha creado nuestra base de datos. Todas las bases de datos SQLite creadas por aplicaciones Android se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón: **/data/data/paquete.java.de.la.aplicacion/databases/nombre_base_datos** En el caso de nuestro ejemplo, la base de datos se almacenaría por tanto en la ruta siguiente: **/data/data/ejemplo.basedatos/databases/DBClientes** Para comprobar que los datos se han grabado podemos acceder de forma remota al emulador a través de su consola de comandos (shell). Pero es mas

sencillo abrir el archivo creado con un bloc de notas y echar un vistazo a su contenido.

Acceso a la BD mediante SQLiteDatabase

La API de Android nos ofrece dos métodos para acceder a los datos. El primero de ellos ya lo hemos visto, se trata de ejecutar sentencias SQL a través de **execSql**. Este método tiene como parámetro una cadena con cualquier instrucción SQL válida. La otra forma es utilizar los métodos específicos **insert()**, **update()** y **delete()** de la clase **SQLiteDatabase**.

Veamos a continuación cada uno de estos métodos.

- **insert()** : recibe tres parámetros `insert(table, nullColumnHack, values)` , el primero es el nombre de la tabla, el segundo se utiliza en caso de que necesitemos insertar valores nulos en la tabla "nullColumnHack" en este caso lo dejaremos pasar ya que no lo vamos a usar y por lo tanto lo ponemos a null y el tercero son los valores del registro a insertar. Los valores a insertar los pasaremos a su vez como elementos de una colección de tipo **ContentValues** . Estos elementos se almacenan como parejas **clave-valor**, donde la clave será el nombre de cada campo y el valor será el dato que se va a insertar.
- **update()** : Prácticamente es igual que el anterior método pero con la excepción de que aquí estamos usando el método `update(table, values, whereClause, whereArgs)` para actualizar/modificar registros de nuestra tabla. Este método nos pide el nombre de la tabla "table", los valores a modificar/actualizar "values" (ContentValues), una condición WHERE "whereClause" que nos sirve para indicarle que valor queremos que actualice y como último parámetro "whereArgs" podemos pasarle los valores nuevos a insertar, en este caso no lo vamos a necesitar por lo tanto lo ponemos a null.
- **delete()** : el método `delete(table, whereClause, whereArgs)` nos pide el nombre de la tabla "table", el registro a borrar "whereClause" que tomaremos como referencia su id y como último parámetro "whereArgs" los valores a borrar.

Insertando con ContentValues

```
val dbClientes = clientes.writableDatabase
val valores = ContentValues()
valores.put("nombre", "Xavi")
valores.put("dni", "22222111")
valores.put("apellidos", "Perez Rico")
dbClientes.update("clientes", valores, "dni=4", null)
dbClientes.close()
```

Borrando con ContentValues

```
val dbClientes = clientes.writableDatabase
if (dbClientes != null) {
    val valores = ContentValues()
    valores.put("nombre", cliente.nombre)
    valores.put("dni", cliente.dni)
    valores.put("apellidos", cliente.apellidos)
    dbClientes.insert("clientes", null, valores)
    dbClientes.close()
}
```


Modificando con ContentValues

```
val dbClientes = clientes.writableDatabase
val arg = arrayOf("1")
dbClientes.delete("clientes", "dni=?", arg)
dbClientes.close()
```

Los valores que hemos dejado anteriormente como null son realmente argumentos que podemos utilizar en la sentencia SQL. Veámoslo con un ejemplo:

```
val dbClientes = clientes.writableDatabase
val valores = ContentValues()
valores.put("nombre", "Carla")
val arg = arrayOf("6", "7")
dbClientes.update("clientes", valores, "dni=? OR dni=?", arg)
dbClientes.close()
```

Donde las `?` indican los emplazamientos de los argumentos.

 **Crea un ejercicio EjercicioResueltoBD en el que pruebes el código visto anteriormente: Creación de la BD, insertar elementos con SQL y modificar, eliminar e insertar mediante ContentValues.**

Recuperación de datos con.rawQuery

Existen dos formas de recuperar información (SELECT de la base de datos), aunque ambas se apoyan en el concepto de **cursor**, que es en definitiva el objeto que recoge los resultados de la consulta. Vamos a completar el ejercicio anterior, para que nos permita ir insertando registros en la BD y visualizándolos en un listView.

La primera forma de obtener datos es utilizar el método `rawQuery()` de la clase `SQLiteDatabase`. Este método recibe como parámetro la sentencia SQL, donde se

indican los campos a recuperar y los criterios de selección.

EjemploBD

dni

nombre

apellidos

ANADIR

MOSTRAR

nombre5
apellidos5
5

nombre0
apellidos0
0

nombre3
apellidos3
3

nombre1
apellidos1
1

```
fun seleccionarDatosSelect(sentencia: String?): Boolean {  
    val listaCliente: ArrayList<Clientes>?  
    val dbClientes = clientes.readableDatabase  
    if (dbClientes != null) {  
        val cursor: Cursor = dbClientes.rawQuery(sentencia, null)  
        listaCliente = getClientes(cursor)  
        dbClientes.close()  
        return if (listaCliente == null) false else true  
    }  
    return false  
}
```

Evidentemente, ahora tendremos que recorrer el cursor y visualizar los datos devueltos. Para ello se dispone de los métodos `moveToFirst()`, `moveToNext()`, `moveToLast()`, `moveToPrevious()`, `isFirst()` y `isLast()`.

Existen métodos específicos para la recuperación de datos `getXXX(indice)`, donde XXX indica el tipo de dato (String, Blob, Float,...) y el parámetro índice permite recuperar la columna indicada en el mismo, teniendo en cuenta que comienza en 0.

El método `getClientes` (implementado por nosotros) es el que nos permite leer los datos existentes en la BD y llevarlos al `arrayList`:

```

fun getClientes(cursor: Cursor): ArrayList<Clientes>? {
    val clientes: ArrayList<Clientes>
    var cliente: Clientes
    cursor.moveToFirst()
    if (!cursor.isAfterLast()) {
        clientes = ArrayList()
        while (!cursor.isAfterLast()) {
            cliente =
                Clientes(cursor.getString(0), cursor.getString(1), cursor.getStrin
            clientes.add(cliente)
            cursor.moveToNext()
        }
        return clientes
    }
    return null
}

```

Recuperación de datos con query

La segunda forma de recuperar información de la BD es utilizar el método `query()`. Recibe como parámetros el nombre de la tabla, un string con los campos a recuperar, un string donde especificar las condiciones del WHERE, otro para los argumentos si los hubiera, otro para el GROUP BY, otro para HAVING y finalmente otro para ORDER BY.


```

fun seleccionarDatosCodigo(
    columnas: Array<String?>?,
    where: String?,
    valores: Array<String?>?,
    orderBy: String?): ArrayList<Clientes>?
{
    var listaCliente: ArrayList<Clientes>? = ArrayList()
    val dbClientes = clientes.readableDatabase
    if (dbClientes != null) {
        val cursor: Cursor =
            dbClientes.query("clientes", columnas, where, valores, null, null,
            listaCliente = getClientes(cursor)
            dbClientes.close()
            return listaCliente
        }
    }
    return null
}

```

Donde la llamada al método podría ser la siguiente:


```
val listaCliente = bdAdapter.seleccionarDatosCodigo(  
    arrayOf("dni", "nombre", "apellidos"),  
    null,  
    null,  
    "apellidos")
```

 **Crea un ejercicio ejemploBD en el que pruebes el código visto anteriormente: Recuperación de los datos con rawQuery y con query. Para ello crea la aplicación como en la imagen, de forma que con un botón se guarde la información en la BD y con el otro se extraiga y se muestre en una lista o en un recycler. Si vais a utilizar una lista como se muestra en la imagen, debereis crear una clase adaptador como la de la siguiente imagen y asociarla a la lista con el setAdapter:**

```
binding.listView.setAdapter(  
    AdaptadorClientes(  
        this@MainActivity,  
        R.layout.list_layout,  
        listaCliente!! ))
```

Clase Adaptador para un ListView:

```
class AdaptadorClientes(var activitycontext: Activity,  
    resource: Int,  
    objects: ArrayList<Clientes>  
    ) :  
    ArrayAdapter<Any>(activitycontext, resource, objects as List<Any>) {  
    var objects: ArrayList<Clientes>  
    override fun getView(position: Int,  
        convertView: View?,  
        parent: ViewGroup): View {  
        var vista: View? = convertView  
        if (vista == null) {  
            val inflater = activitycontext.layoutInflater  
            vista = inflater.inflate(R.layout.list_layout, null)  
            (vista?.findViewById(R.id.apellidolist) as TextView)  
                .setText(objects[position].apellidos)  
            (vista?.findViewById(R.id.nombrelist) as TextView)  
                .setText(objects[position].nombre)  
            (vista?.findViewById(R.id.dnिलist) as TextView)  
                .setText(objects[position].dni)  
        }  
        return vista  
    }  
  
    init {  
        this.objects = objects  
    }  
}
```

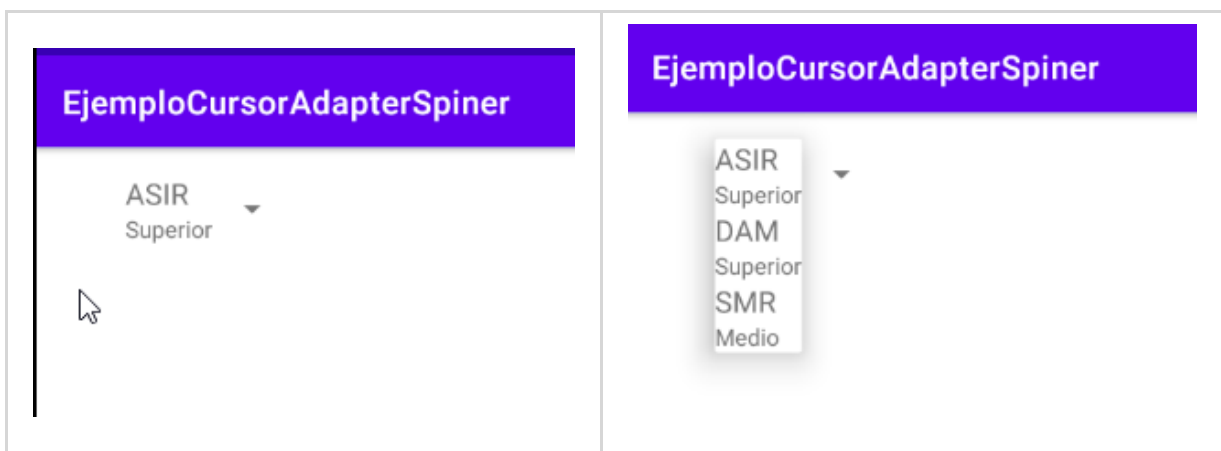
 **Ejercicio Propuesto Recorrer BD**

Uso de SimpleCursorAdapter

Se utiliza para mapear las columnas de un cursor abierto en una base de datos a los diferentes elementos visuales contenidos en el control de selección. Así se evita el volcado de todos los elementos descargados de la BD a una colección, con el ahorro de memoria que conlleva.

Nos encontramos con dos casos: cuando usamos **Spinner** o **ListView** sencillos el adaptador que gestiona este volcado ya está creado, si usamos **RecyclerView** tendremos que crear y gestionar el adaptador.

Vamos a programar un ejemplo para practicar este concepto, será una sencilla actividad con un **Spinner** que mostrará los ciclos informáticos con la categoría a la que pertenecen:



Para ello nos creamos la base de datos e insertamos elementos como ya hemos visto hasta el momento:

```

class DBAdapter(context: Context) {
    private var ohCategoria: OHCategoria
    init {
        ohCategoria = OHCategoria(context, "BBDCategorias", null, 1)
    }
    fun insertarDatosCodigo() {
        val sqLiteDatabase = ohCategoria.writableDatabase
        if (sqLiteDatabase != null) {
            val valores = ContentValues()
            valores.put("nombre", "ASIR")
            valores.put("cate", "Superior")
            valores.put("idcategoria", 1)
            sqLiteDatabase.insert("categoria", null, valores)
            valores.put("nombre", "DAM")
            valores.put("cate", "Superior")
            valores.put("idcategoria", 2)
            sqLiteDatabase.insert("categoria", null, valores)
            valores.put("nombre", "SMR")
            valores.put("cate", "Medio")
            valores.put("idcategoria", 3)
            sqLiteDatabase.insert("categoria", null, valores)
            sqLiteDatabase.close()
        }
    }
    fun leerDatos():Cursor?
    {
        val sqLiteDatabase = ohCategoria.readableDatabase
        if (sqLiteDatabase != null) {
            return sqLiteDatabase.rawQuery(
                "select idcategoria as _id, nombre, cate from categoria",
                null )
        }
        return null
    }

    inner class OHCategoria(
        context: Context?, name: String?,
        factory: SQLiteDatabase.CursorFactory?,
        version: Int):QLiteOpenHelper(context, name, factory, version)
    {
        var cadena =
            "create table if not exists categoria(idcategoria
            INTEGER PRIMARY KEY NOT NULL, nombre TEXT, cate TEXT);"
        override fun onCreate(db: SQLiteDatabase) {
            db.execSQL(cadena)
        }
        override fun onUpgrade(db: SQLiteDatabase,
                                oldVersion: Int, newVersion: Int) {}
    }
}

```

Posteriormente pasamos a crear el `cursorAdapter`, pero para ello vamos a usar un `Spinner` para mostrar la información, lo incluimos en el layout principal.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val from = arrayOf("nombre", "cate")
    val to = intArrayOf(R.id.ciclo, R.id.cate)
    setContentView(R.layout.activity_my)
    val dbAdapter=DBAdapter(this)
    dbAdapter.insertarDatosCodigo()
    val desplegable = findViewById<Spinner>(R.id.spinner)
    9 val cursor=dbAdapter.leerDatos()
    10 val mAdapter = SimpleCursorAdapter(this, R.layout.spinner_layout,
                                         cursor, from, to, 0x0)
    desplegable.setAdapter(mAdapter)
}
```

📌 en la **Línea 9** se devuelve el cursor con la selección de los datos de la BD (cuidado!! no se deberá cerrar el cursor). **Línea 19** para el funcionamiento del **SimpleCursorAdapter** , es necesario indicar en su definición los siguientes elementos: el contexto, el layout sobre el que se va a definir la vista, el cursor que va a ser origen de los datos, lista con el nombre de las columnas que se quiere enlazar, una lista con los id de las vistas donde se van a mostrar los campos (el orden es importante) y un flag (0 por defecto)

🖋 **Prueba la aplicación del ejemplo anterior**

Cursores con RecyclerView

Como era de esperar, no se puede aplicar directamente un adaptador de tipo

`SimpleCursorAdapter` a un tipo `RecyclerView`. Si queremos aprovechar los beneficios de los cursores en los recylcers, tendremos que fabricarlos a medida. Los pasos son los siguientes:

1. Crear una clase **Abstracta** base que derive de **RecyclerView.Adapter** y en la que implementaremos los métodos sobrescritos derivados de `RecyclerView.Adapter`:

```
abstract class CursorRecyclerViewAdapter(cursor: Cursor) :  
    RecyclerView.Adapter<RecyclerView.ViewHolder>() {  
    var mCursor: Cursor  
    override fun onBindViewHolder(holder: RecyclerView.ViewHolder,  
        position: Int)  
    {  
        checkNotNull(mCursor) { "ERROR, cursos vacio" }  
        check(mCursor.moveToPosition(position)) { "ERROR, no se puede" +  
            " encontrar la posicion: $position" }  
        onBindViewHolder(holder, mCursor)  
    }  
    12 abstract fun onBindViewHolder(holder: RecyclerView.ViewHolder,  
        cursor: Cursor)  
    override fun getItemCount(): Int  
    {  
        16 return if (mCursor != null) mCursor.getCount()  
            else 0  
    }  
    init {  
        mCursor = cursor  
    }  
}
```

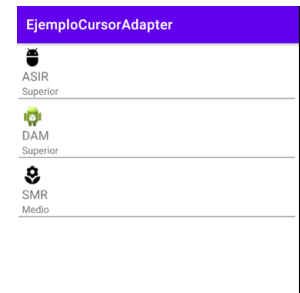
✚ Al heredar de `RecyclerView.Adapter` habrá que implementar los métodos `onBindViewHolder()` al que le llega la posición de la línea del recycler que está activa en ese momento y `getItemCount()` que tendrá que devolver los elementos que tiene el cursor **Línea 16**, le llega en el constructor. En el método `onBindViewHolder` se comprobará que el cursor no está vacío y que se puede acceder a la posición que llega como parámetro **Línea 8**, y en ese caso se llamará al método `onBindViewHolder(holder: RecyclerView.ViewHolder, cursor: Cursor)` abstracto que debemos crear en esta clase **Línea 12**, pero esta vez con un `Cursor` como parámetro.

2. Ahora tendremos que crear nuestra clase RecyclerView derivada de la anterior.

```
class RecyclerViewAdapter(c: Cursor) : CursorRecyclerViewAdapter(c) {
    override fun onCreateViewHolder(parent: ViewGroup,
                                    viewType: Int): RecyclerView.ViewHolder
    {
        val v: View = LayoutInflater.from(parent.context)
            .inflate(R.layout.recycler_layout, parent, false)
        return SimpleViewHolder(v)
    }
    override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
                                   cursor: Cursor)
    {
        (holder as SimpleViewHolder).bind(cursor)
    }
    internal inner class SimpleViewHolder(itemView: View) :
        RecyclerView.ViewHolder(itemView)
    {
        var nombre: TextView
        var cate: TextView
        var imagen: ImageView
        fun bind(dato: Cursor) {
            nombre.setText(dato.getString(0))
            cate.setText(dato.getString(1))
            val theImage: Bitmap = MainActivity.
                convertirStringBitmap(dato.getString(3))
            imagen.setImageBitmap(theImage)
        }
        init {
            nombre = itemView.findViewById(R.id.ciclo)
            cate = itemView.findViewById(R.id.cate)
            imagen = itemView.findViewById(R.id.imagen) as ImageView
        }
    }
}
```

✚ A esta clase le llegará el cursor por el constructor que a su vez será pasado a la clase padre. Al heredar de la clase abstracta, tendremos que implementar su método abstracto y será ahí donde aprovecharemos que el cursor se ha posicionado en el elemento correcto de la BD, para llamar al método **bind(cursor)** **Línea 12** del **Holder** , con ese estado del cursor. En el **Holder** trataremos los elementos que sacamos del cursor de forma correcta.

3. Como esta App es un poco diferente a la anterior, se han incluido imágenes para dar más funcionalidad, hay dos métodos que nos permiten pasar imágenes de Bitmap a String y viceversa para poderlas guardar en la BD a través del cursor (En la BD se guardarán como Blob). Están localizados en MyActivity y son estáticos para poder acceder a ellos sin necesidad de objeto.



```
companion object
{
    fun convertirStringBitmap(imagen: String?): Bitmap {
        val decodedString: ByteArray = Base64.decode(imagen, Base64.DEFAULT)
        return BitmapFactory.decodeByteArray(decodedString, 0, decodedString.size)
    }

    fun ConvertirImagenString(bitmap: Bitmap): String {
        val stream = ByteArrayOutputStream()
        bitmap.compress(Bitmap.CompressFormat.PNG, 90, stream)
        val byte_arr: ByteArray = stream.toByteArray()
        return Base64.encodeToString(byte_arr, Base64.DEFAULT)
    }
}
```

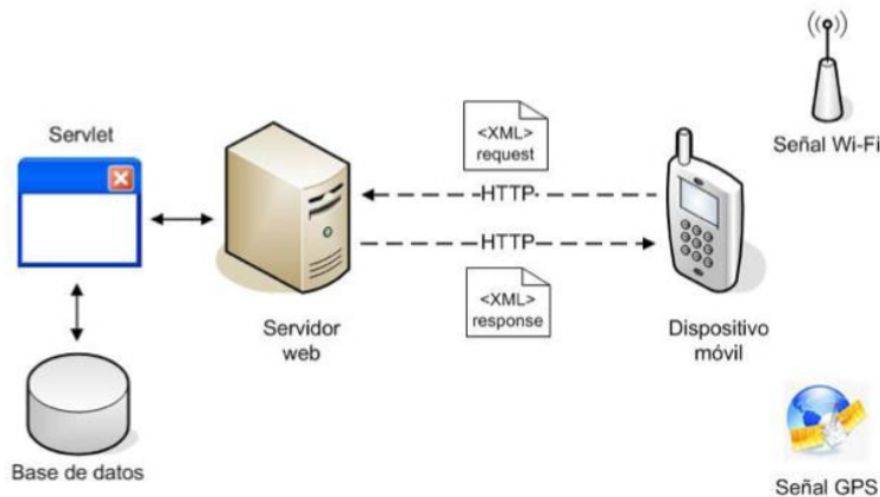
 **Prueba la aplicación del ejemplo anterior**

 **Ejercicio propuesto cursor con recycler**

Acceso a bases de datos remotas

Introducción

Para obtener y guardar información de una base de datos remota, es necesario conectarse a un servidor donde se encontrará la BBDD. El esquema de funcionamiento lo podemos ver en la siguiente imagen:



En el servidor funcionan en realidad tres componentes básicos:

- Una base de datos, que almacena toda la información de los usuarios. Para la BBDD se puede utilizar MySQL, que es de licencia gratuita.



- Un servlet , que atiende la petición recibida, la procesa y envía la respuesta correspondiente. Un servlet no es más que un componente Java, generalmente pequeño e independiente de la plataforma.
- Un servidor web , donde reside y se ejecuta el servlet , y que permanece a la espera de conexiones HTTP entrantes.

Los formatos más utilizados para compartir información mediante estos servicios web son XML (y otros derivados) y JSON .

¿Qué es JSON?

JSON (Javascript Objec t Notation) es un formato ligero de intercambio de datos entre clientes y servidores, basado en la sintaxis de Javascript para representar estructuras en forma organizada. Es un formato en texto plano independiente de todo lenguaje de programación, es más, soporta el intercambio de datos en gran variedad de lenguajes de programación como PHP Python C++ C# Java y Ruby.

XML también puede usarse para el intercambio, pero debido a que su definición genera un DOM , el parseo se vuelve extenso y pesado. Además de ello XML debe usar Xpath para especificar rutas de elementos y atributos, por lo que demora la reconstrucción de la petición. En cambio JSON no requiere restricciones adicionales, simplemente se obtiene el texto plano y el engine de Javascript en los navegadores hace el trabajo de parsing sin ninguna complicación.

Tipos de datos en JSON

Similar a la estructuración de datos primitivos y complejos en los lenguajes de programación, JSON establece varios tipos de datos: cadenas, números, booleanos, arrays, objetos y valores null. El propósito es crear objetos que contengan varios atributos compuestos como pares clave valor. Donde la clave es un nombre que identifique el uso del valor que lo acompaña. Veamos un ejemplo:

```
{
  "id": 101,
  "Nombre": "Carlos",
  "EstaActivo": true,
  "Notas": [2.3, 4.3, 5.0]
}
```

La anterior estructura es un objeto JSON compuesto por los datos de un estudiante. Los objetos JSON contienen sus atributos entre llaves {}, al igual que un bloque de código en Javascript, donde cada atributo debe ir separado por coma , para diferenciar cada par.

La sintaxis de los pares debe contener dos puntos : para dividir la clave del valor. El nombre del par debe tratarse como cadena y añadirle comillas dobles.

Si notas, este ejemplo trae un ejemplo de cada tipo de dato:

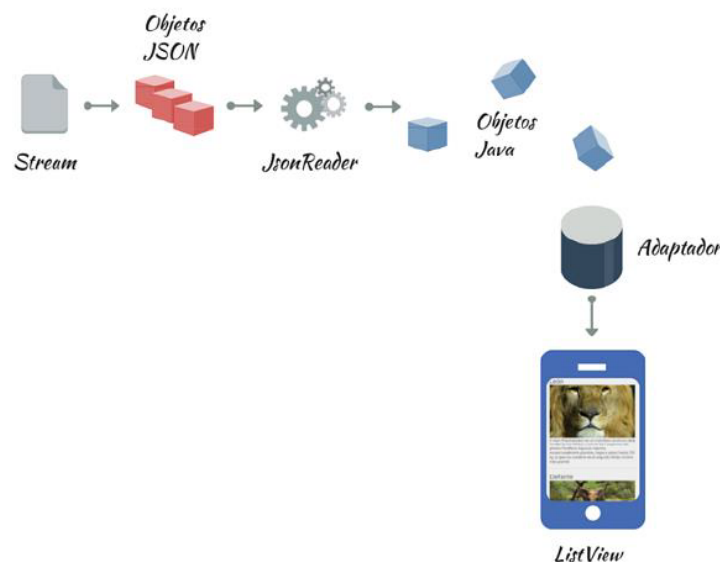
- El *id* es de tipo entero, ya que contiene un número que representa el código del estudiante.

- El *Nombre* es un string. Usa comillas dobles para identificarlas.
- *EstaActivo* es un tipo booleano que representa si el estudiante se encuentra en la institución educativa o no. Usa las palabras reservadas true y false para declarar el valor.
- *Notas* es un arreglo de números reales. El conjunto de sus elementos debes incluirlos dentro de corchetes `[]` y separarlos por coma.

La idea es crear un mecanismo que permita recibir la información que contiene la base de datos externa en formato JSON hacia la aplicación. Con ello se parseará cada elemento y será interpretado en forma de objeto Java para integrar correctamente el aspecto en la interfaz de usuario.

La clase **JsonObject** de la librería **org.json.JSONObject**, puede interpretar datos con formato JSON y parsearlos a objetos Java o a la inversa.

Veamos una ilustración que muestra el proceso de parseo que será estudiado:



- Como puedes observar el origen de los datos es un servidor externo o hosting que hayas contratado como proveedor para tus servicios web. La aplicación web que realiza la gestión de encriptación de los datos a formato `JSON` puede ser `PHP`, `JavaScript`, `ASP.NET`, etc..
- Tu aplicación Android a través de un cliente realiza una petición a la dirección URL del recurso con el fin de obtener los datos. Ese flujo entrante debe interpretarse con ayuda de un parser personalizado que implementaran las clases que se utilizan para trabajar con `JSON`.
- El resultado final es un conjunto de datos adaptable al `API` de Android. Dependiendo de tus necesidades, puedes convertirlos en una lista de objetos estructurados que alimenten un adaptador que pueble un `ListView` o simplemente actualizar la base de datos local de tu aplicación en `SQLite`.

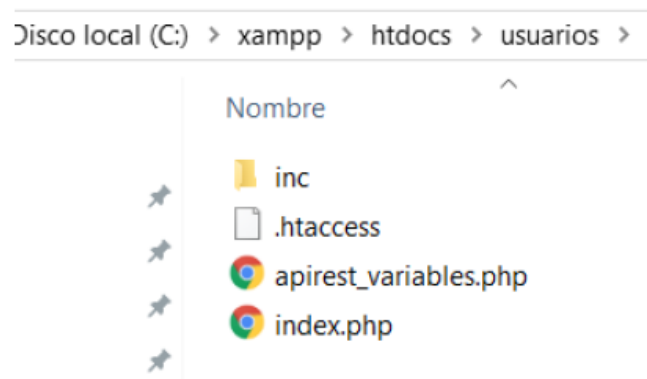
Acceso a bases de datos con ApiRest y Retrofit

Retrofit es un cliente *REST* para Android y Java, desarrollado por Square, muy simple y fácil de aprender. Permite hacer peticiones **GET**, **POST**, **PUT**, **PATCH**, **DELETE** y **HEAD**; gestionar diferentes tipos de parámetros y parsear automáticamente la respuesta a un POJO (Plain Old Java Object). Vamos a utilizar esta librería por su facilidad de manejo.

Definición del Servicio Rest

Como es de suponer, para poder acceder a un servicio Rest desde una de nuestras aplicaciones, este debe de haber sido creado con anterioridad y alojado en un Hosting adecuado.

La parte de creación ya sea con PHP, Java, o cualquiera de los otros lenguajes que lo permiten no corresponde a nuestra asignatura, por lo que la pasaremos por alto y usaremos un ApiRest general que se ha proporcionado en el módulo de Acceso a Datos y que con unas pequeñas modificaciones es valido para la mayoría de casos. La carpeta con el ApiRest está compuesta de los siguientes archivos:



La configuración de nuestro API consta de dos archivos:

1. **.htaccess** En este archivo se configuran las reglas de acceso, sobre una ruta que le indiquemos en RewriteBase (aquí es donde tendremos que añadir la carpeta en la que habremos alojado nuestra API, comenzando y acabando con `/`).
2. **apirest_variables.php**, en este archivo se definen los datos de conexión a la base de datos. se indican las tablas que tiene esta y el nombre del identificador de cada una de ellas.

<pre> .htaccess RewriteEngine On RewriteBase "/"usuarios/" RewriteRule ^([a-zA-Z-]*)\$ index.php?action=\$1&{QUERY_STRING} RewriteRule ^([a-zA-Z-]*)/([a-zA-Z0-9-]*)\$ index.php?action=\$1&value=\$2&{Q RewriteRule ^([a-zA-Z-]*)/([a-zA-Z-]*)/([a-zA-Z0-9-]*)\$ index.php?action=\$1 </pre>	<pre> apiREST_variables.php <?php // CONFIGURACIÓN BASE DE DATOS MYSQL \$servername = "127.0.0.1"; \$username = "root"; \$password = ""; // BASE DE DATOS \$dbname = "usuariosmensajes"; // TABLAS Y SU CLAVE \$tablas = array(); \$tablas["mensajes"]="_id"; \$tablas["usuarios"]="_id"; </pre>
--	---

El resto de archivos del ApiREst no tendrán que modificarse, ya que está construida de forma genérica con las necesidades más comunes para estos casos. Tendremos que crear la BD y alojar el ApiRest de forma local o en la nube, usando los conocimientos que se tienen del módulo de Acceso a Datos.

Vamos a suponer un ejemplo muy sencillo de una base de datos Usuarios en el que tendremos solamente una tabla Usuarios con dos campos de tipo String (nick y nombre). La BD la habremos construido en el servidor con antelación (en este caso con tabla usuarios de tres campos, nick y nombre de tipo cadena y _id de tipo numérico autoincrementable como clave). También crearemos una aplicación con dos campos de texto que nos permita insertar los datos del usuario y un botón flotante de añadir, como se ve en la imagen siguiente:

Consumo de un Servicio Rest desde Android

Existen diferentes librerías que nos permitirían consumir los servicios desde la App de Android, pero dada su facilidad vamos a utilizar las librerías: **Retrofit2** y **Gson**. Retrofit la utilizaremos para hacer peticiones y procesar las respuestas del APIRest, mientras que con Gson transformaremos los datos de JSON a los propios que utilice la aplicación.

Para ello añadiremos las siguientes líneas en el build.gradle de la app, y no olvides incluir permisos de internet:

```
implementation "com.squareup.retrofit2:converter-gson:2.9.0"
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
```

Podemos decir que los pasos a seguir serán los siguientes:

1. Creación de un builder de Retrofit. Para poder utilizar Retrofit necesitas crear un builder que te servirá para comunicarte con la API elegida. En la imagen se ha utilizado un método para encapsular el código, aunque no es necesario.

```
private fun crearRetrofit(): ProveedorServicio {
    //val url = "http://10.0.2.2/usuarios/" //para el AVD de android
    val url="http://xusa.iesdoctorbalmis.info/usuarios/" //para servidor del ins
    val retrofit = Retrofit.Builder()
        .baseUrl(url)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
    return retrofit.create(ProveedorServicio::class.java)
}
```

Si te fijas en la segunda línea verás que hay que escribir la url base de la API a la que harás las peticiones. La obtienes de la documentación proporcionada por el creador de la API.

En la siguiente línea hay que indicar la forma de convertir los objetos de la API a los de tu aplicación y viceversa, aquí es donde especificas que vas a utilizar Gson.

Este builder lo necesitarás para hacer las llamadas a la API, así que procura que sea accesible.

2. Creación de las clases Pojo que le servirá al Gson para parsear los resultados. Pero primero necesitas conocer la estructura del Json que va a devolverte la API, ya que no hay un patrón establecido. Para conocer la estructura previamente, se puede utilizar PostMan y realizar las diferentes peticiones (GET, POST, PUT, etc) desde este. Existen aplicaciones o webs que facilitan la creación de la clase Pojo a partir de un JSON, como por ejemplo: [<http://www.jsonschema2pojo.org/>]

```
class RespuestaJSON {
    var respuesta = 0
    var metodo: String? = null
    var tabla: String? = null
    var mensaje: String? = null
    var sqlQuery: String? = null
    var sqlError: String? = null
}
```

Después tienes que crear la estructura de clases para almacenar la información que te resulte útil. Clase Usuario en este ejemplo.

```
class Usuarios(var nick: String, var nombre: String, var _id: Int = 0)
```

3. Otro elemento imprescindible, es la gestión de los servicios que se quieran utilizar. Para cada uno de ellos se tendrá que hacer una petición a la API. Necesitaremos crear una interfaz con todos los servicios que quieras utilizar. Aquí tienes unos ejemplos:

```
interface ProveedorServicio {
    @GET("usuarios")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun usuarios(): Response<List<Usuarios>>

    @GET("mensajes")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun mensajes(): Response<List<Mensaje>>

    @GET("mensajes/{nick}")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun getUsuario(@Path("nick") nick: String): Response<List<Usuarios>>

    @POST("usuarios")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun insertarUsuario(@Body usuarios: Usuarios): Response<RespuestaJSon>

    @POST("mensajes")
    @Headers("Accept: application/json", "Content-Type: application/json")
    suspend fun insertarMensaje(@Body mensaje: Mensaje): Response<RespuestaJSon>
}
```

En el código de arriba hay dos servicios dos son de tipo **GET** sin palabra clave y que servirán para obtener todos los usuarios y todos los mensajes. Luego se tiene otro servicio **GET** con palabra clave (Nick) que servirá para seleccionar los mensajes de un determinado Nick. Y luego dos **POST** a los que se les pasa en el Body los datos a añadir.

Usando **Corrutinas** se realizará la llamada a la API, por lo que los métodos de la interfaz deben ser de tipo **suspend**. En cada método se indica el tipo de dato que espera obtener, en estos ejemplos la respuesta puede ser o una lista del tipo de objeto de la petición o un tipo RespuestaJSo, que tendrá todos los posibles miembros que nos podría dar alguna de las llamadas invocadas.

Las palabras seguidas del símbolo **@** dan funcionalidad a los servicios, haciéndolos dinámicos y reutilizables, para más información [<http://square.github.io/retrofit/>]

💡 Las que gestiona **Retrofit** para invocar la url de la petición son **@GET**, **@POST**, **@PUT** y **@DELETE** . El parámetro corresponderá con la url de la petición.

En el builder de Retrofit se incluye la url base del api terminada con **/** , que unida con el parámetro del servicio crearán la url completa de la petición:

@GET -> http://10.0.2.2/usuarios/usuario

- **@Header** y **@Headers** . Se usan para especificar los valores que van en la sección *header* de la petición, como por ejemplo en que formato van a ser enviados y recibidos los datos.
- **@Path** . Sirve para incluir un identificador en la url de la petición, para obtener información sobre algo específico. El atributo en el método de llamada que sea precedido por *@Path*, sustituirá al identificador entre llaves de la ruta que tenga el mismo nombre.
- **@Fields** . Nos permite pasar variables básicas en las peticiones *Post* y *Put*.
- **@Body** . Es equivalente a Fields pero para variables objeto.
- **@Query** . Se usa cuando la petición va a necesitar parámetros (los valores que van después del **?** en una url).

4. Pedir datos a la Api sería el último paso a realizar. Retrofit nos da la opción de realizarlo de manera síncrona o asíncrona. Vamos a aprovechar nuestros conocimientos de corrutinas para lanzar la petición en segundo plano de forma sencilla.


```

        private fun anyadirUsuario(usuarios: Usuarios) {
            val context=this
            var salida:String?
4            val proveedorServicios: ProveedorSersvicio = crearRetrofit()
5            CoroutineScope(Dispatchers.IO).launch {
                val response = proveedorServicios.insertarUsuario(usuarios)
                if (response.isSuccessful) {
8                    val usuariosResponse = response.body()

                    if (usuariosResponse != null) salida=usuariosResponse.mensaj
                    else salida=response.message()                }
                else {
                    Log.e("Error", response.errorBody().toString())
                    salida=response.errorBody().toString()
                }
                withContext(Dispatchers.Main) { Toast.makeText(context, salida,
                    if (espera != null) espera?.hide()
                    limpiaControl())}
19            }
        }
    }

```

✦ **Línea 4** llamamos al método `crearRetrofit`, que es el que nos devuelve un proveedor de servicios del tipo de interface que hemos creado y nos enlazaré con la url del servidor y con el GSON. Línea 5 - 13 se lanza la corrutina con la petición deseada, cuando se obtenga respuesta se filtra para ver si ha sido correcta y recuperar los datos necesarios, `response.body()`, en caso contrario se lanza mensaje de error.

✍ **EjercicioPropuestoBDExternas**

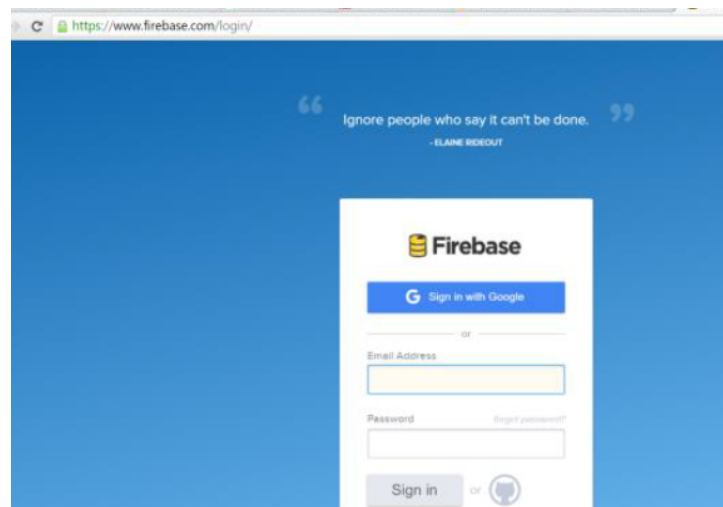
✍ **EjercicioPropuestoAgenda**

Acceso a Base de Datos con FIREBASE

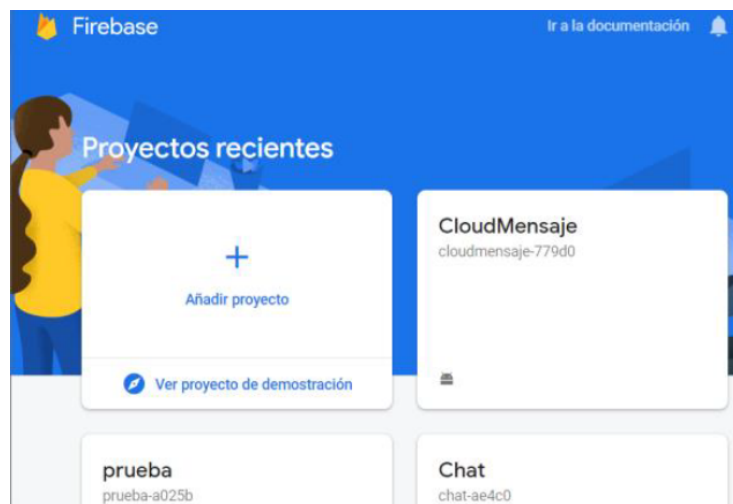
Firebase es una plataforma de backend para construir aplicaciones móviles y web, se encarga del manejo de la infraestructura permitiendo que el desarrollador se enfoque en otros aspectos de la aplicación. Entre sus características se incluye base de datos de tiempo real, autenticación de usuarios y almacenamiento (hosting) estático. La idea de usar Firebase es no tener que escribir código del lado del servidor y aprovechar al máximo las características que nos provee la plataforma. Para utilizar Firebase con Android disponemos de un SDK, lo que nos permitirá integrarlo fácilmente a nuestra aplicación. Para este ejemplo vamos a construir un listado de cosas por hacer usando la base de datos de tiempo real de Firebase como backend y autenticación con email/password.

Creando un app de Firebase

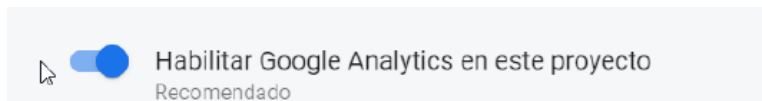
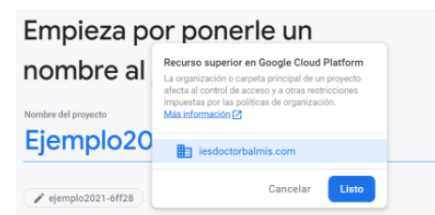
El primer paso es asociar nuestra cuenta de Firebase a una de nuestras cuentas de correo electrónico:



Una vez está lista veremos la pantalla donde es posible visualizar nuestros proyectos o crear nuevos.



Para crear un nuevo proyecto solamente tendremos que introducir un nombre y la organización o carpeta principal del proyecto. Google nos pedirá si queremos habilitar Google Analytics en el proyecto, lo dejaremos habilitado.



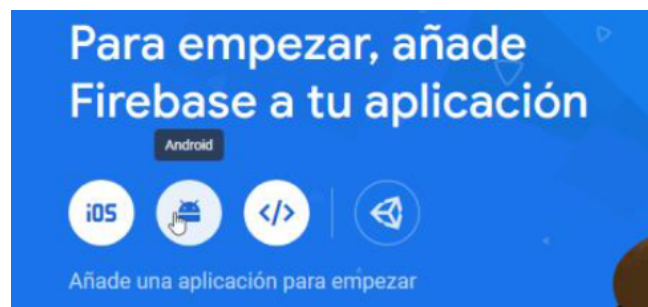
Y lo asociaremos a la cuenta por defecto. Una vez realizados estos pasos, se creará el proyecto de firebase.

Posteriormente nos aparecerá la pantalla que nos permitirá añadir la aplicación a nuestro proyecto. Cada proyecto estará asociado a una o más aplicaciones, por lo que antes de crear el proyecto necesitaremos crear la aplicación o decidir el espacio de nombres de esta.

Vamos a crear una aplicación sencilla con el siguiente aspecto, que nos permita entender el funcionamiento de acceso a Firebase para enviar y recibir datos sin autenticación.



Cuando el usuario meta la información en los TextView y pulse el botón añadir, la información se guarda en Firebase y cuando se modifique esta desde Firebase, se mostrará un mensaje con el texto modificado en el Txt de la aplicación.



Una vez tengamos el espacio de nombre, podremos seguir con los pasos que nos pide Firebase para crear la aplicación. En nuestro caso no será necesario introducir la firma de certificación.

Una captura de pantalla de la interfaz de Firebase para "Añadir Firebase a una aplicación de Android". El formulario tiene un título "1 Registrar la aplicación" y tres campos de entrada: "Nombre del paquete de Android" con el valor "com.tema12.cloudfirestore.ejemplousuarios", "Apodo de la aplicación (opcional)" con el valor "EjemploUsuariosCloudFirestore20", y "Certificado de firma de depuración SHA-1 (opcional)" que está vacío. Debajo de los campos hay un texto explicativo sobre la obligatoriedad del certificado para ciertos servicios. Al final del formulario hay un botón azul que dice "Siguiente".

Justo en el momento que termina de crearse la aplicación, se descargara un archivo JSON que deberemos copiar en nuestro proyecto Android. Solo tendremos que seguir las instrucciones que proporciona muy claramente la página Web de Firebase (copiar el archivo y añadir las líneas que indican que vamos a usar servicios de google en los build.gradle de la app y del proyecto).

1. En el proyecto

```
dependencies {  
    classpath "com.android.tools.build:gradle:7.0.3"  
    classpath "com.google.gms:google-services:4.3.10" //añadir esta línea  
    ...  
}
```

2. En la APP

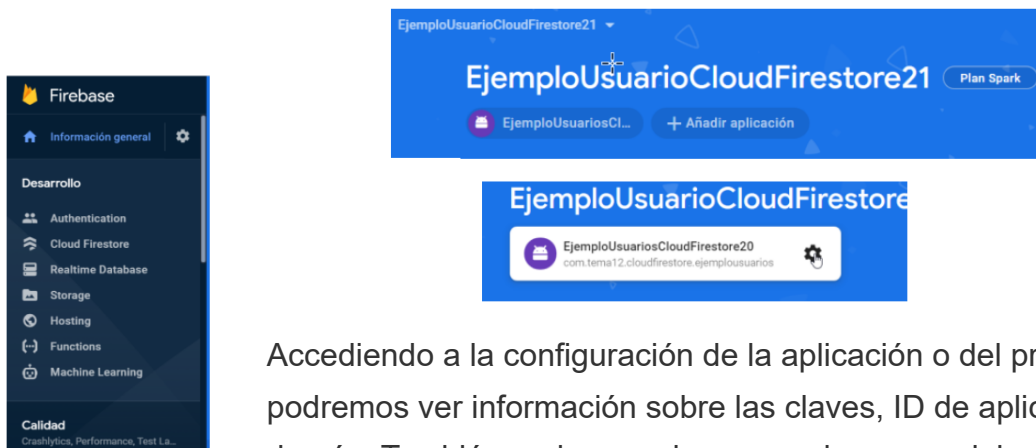
```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
    id 'com.google.gms.google-services' // Google Services plugin  
}  
  
dependencies {  
    ...  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    //añadir las siguientes dos líneas  
    implementation platform('com.google.firebase:firebase-bom:28.4.1')  
    implementation 'com.google.firebase:firebase-analytics'  
    ...  
}
```

La dependencia de firebase bom permitirá evitar añadir la versión al resto de dependencias de Firebase que incluyamos en nuestro proyecto, facilitando mucho el trabajo.

Muy importante no olvidar añadir en el SDK, el servicio de Google Play

<input type="checkbox"/> Google Play Licensing Library	1	Not installed
<input checked="" type="checkbox"/> Google Play services	49	Installed
<input checked="" type="checkbox"/> Google USB Driver	12	Not installed
<input type="checkbox"/> Google Web Driver	2	Not installed

En el modo consola podremos acceder a todos nuestros proyectos, y podemos comprobar la aplicación o aplicaciones añadidas sobre un proyecto (podemos añadir más de una).



Accediendo a la configuración de la aplicación o del proyecto, podremos ver información sobre las claves, ID de aplicación y demás. También podremos descargar de nuevo el .json de configuración de los servicios.

Todas las aplicaciones por defecto se encuentran en modo desarrollo y bajo el plan gratis que soporta hasta 100 conexiones concurrentes, 1GB de almacenamiento y 10GB de transferencia en el backend. Luego de creada el app nos dirigimos a ver sus detalles, podemos entrar en diversas pestañas que nos permitirán trabajar con la

APP, pero a nosotros nos interesa la BD. Firebase ofrece dos soluciones de bases de datos en la nube y accesibles para los clientes que admiten sincronización en tiempo real:

- **Firestore DataBase** en ocasiones también nombrada Cloud Firestore, es la base de datos más reciente de Firebase. Aprovecha lo mejor de Realtime Database con un modelo de datos nuevo, permite consultas más ricas y rápidas, y el escalamiento se ajusta a un nivel más alto que Realtime Database.
- **Realtime Database** es la base de datos original de Firebase. Es una solución eficiente y de baja latencia destinada a las apps para dispositivos móviles que necesitan estados sincronizados entre los clientes en tiempo real. Es similar a la anterior en su uso, por lo que no la explicaremos.

Realtime Database	Cloud Firestore
Almacena datos como un gran árbol JSON	Almacena datos como colecciones de documentos
<ul style="list-style-type: none">• Los datos simples son muy fáciles de almacenar.• Los datos complejos y jerárquicos son más difíciles de organizar a gran escala.	<ul style="list-style-type: none">• Los datos simples son fáciles de almacenar en documentos, que son muy similares a JSON.• Los datos complejos y jerárquicos son más fáciles de organizar a escala, con subcolecciones dentro de los documentos.• Necesita menos desnormalización y compactación de datos.

Firestore DataBase

Base de datos NoSQL flexible, escalable y en la nube a fin de almacenar y sincronizar datos para la programación en el lado del cliente y del servidor. Al igual que [Firebase Realtime Database](#) mantiene los datos sincronizados entre apps cliente a través de agentes de escucha en tiempo real. El modelo de datos de [Firestore DataBase](#) admite estructuras de datos flexibles y jerárquicas. Almacena los datos en documentos organizados en colecciones, los documentos pueden contener objetos anidados complejos, además de subcolecciones. Está optimizada para almacenar grandes colecciones de documentos pequeños. El modelo de datos está formado por documentos, cada documento contiene un conjunto de pares clave-valor. Todos los documentos se deben almacenar en colecciones, los documentos pueden contener subcolecciones y objetos anidados, y ambos pueden incluir campos primitivos como strings o tipos de objetos complejos como listas. Las colecciones y los documentos se crean de manera implícita en **Firestore DataBase**. Solo debes asignar datos a un documento dentro de una colección. Si la colección o el

documento no existen, **Firestore DataBase** los crea. Cada documento está identificado por una clave única, que puede generarse automáticamente o que se puede añadir a la vez que el documento:

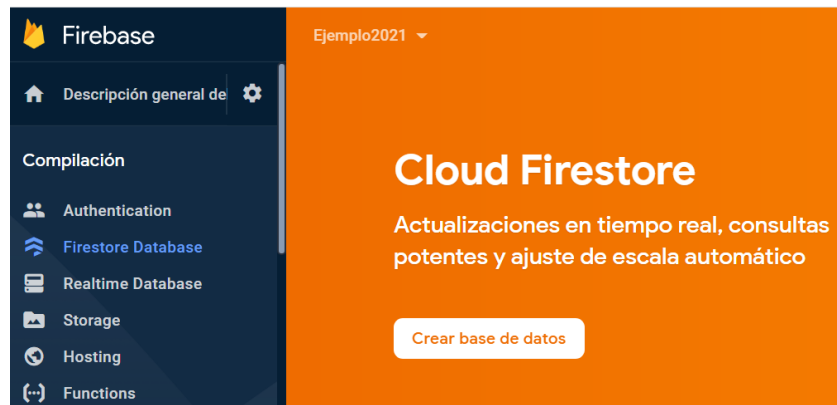
items	U8jXrWZQeWCGBcy7C7fd	items	dH6paTTRgtj61iI5etUU
+ Añadir documento	+ Iniciar colección	+ Añadir documento	+ Iniciar colección
4ASan8jPPxLWfY62ytEO	+ Añadir campo	4ASan8jPPxLWfY62ytEO	+ Añadir campo
U8jXrWZQeWCGBcy7C7fd	msg: "El primer mensaje"	U8jXrWZQeWCGBcy7C7fd	key: null
dH6paTTRgtj61iI5etUU	user: "xusa"	dH6paTTRgtj61iI5etUU	msg: "Como va todo?"
			user: "xusa33"

Son muy similares a JSON, de hecho, básicamente son JSON. Existen algunas diferencias, por ejemplo: los documentos admiten **tipos de datos adicionales** y su tamaño se limita a 1 MB, pero en general, puedes tratar los documentos como registros JSON livianos. Los documentos viven en colecciones, que simplemente son contenedores de documentos. Por ejemplo, podrías tener una colección llamada users con los distintos usuarios de tu app, en la que haya un documento que represente a cada uno:



Usando el ejercicio del que hemos creado el layout de inicio, vamos a añadir el código necesario para que nos permita interactuar con Firebase. La aplicación guardará usuarios con contraseña en la base de datos con **Firestore Cloud Firestone** y además permitirá la autenticación con **Firestore Auth**.

En el proyecto de Firebase que acabamos de crear, añadiremos una base de datos **Firestore DataBase**.



Deberemos seleccionar la ubicación de nuestra Base de Datos, podemos seleccionar ZÚRICH(europe-west6). Un elemento importante y que todavía no hemos mencionado, son la Reglas. La Firebase Cloud Firestore proporciona un lenguaje de reglas flexibles basadas en expresiones y sintaxis similar a la de JavaScript <https://firebase.google.com/docs/firestore/security/get-started?authuser=0>, que permite definir fácilmente la manera en que tus datos deben estructurarse e indexarse, y el momento en que pueden someterse a lectura y escritura. Estas reglas las podremos configurar desde la pestaña RULES de nuestra BD,



Por defecto vendrían configuradas para modo producción, aunque para iniciarse podemos seleccionar modo de prueba (dura 30 días).




```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
        request.time < timestamp.date(2021, 1, 1);
    }
  }
}
```

Si nos hubiésemos confundido o quisiéramos cambiar las reglas, habría que cambiar su valor y sobre todo **No olvidar Publicar**.

Antes de comenzar con el código de acceso a la BD, tendremos que añadir la dependencia que nos permitirá hacerlo:

```
implementation 'com.google.firebase:firebase-firestore'
```

Para referenciar a nuestra base de datos solamente tendremos que crear un objeto de tipo **Firestore**. El archivo que descargamos al enlazar la app con el proyecto, es el que se encarga de todo el trabajo interno para la conexión:

```
val firestore = FirebaseFirestore.getInstance()
```

Para hacer referencia a una colección existente o añadirla en caso de no existir, utilizaremos el método **collection** con el nombre de la colección como argumento.

```
firestore.collection("Usuarios")
```

Este método devuelve una referencia a la colección seleccionada, y con él podremos añadir nuevos elementos a ella (la colección se creará al añadir el primer documento, en caso de haber sido creada anteriormente hará la referencia solamente).

Para añadir un objeto (documento) a la colección lo podremos hacer de dos maneras:

1. Dejando que la plataforma genere una clave aleatoria, para eso utilizaremos el método `add`.

```
firestore.collection("Usuarios").add(Usuario("Pepe", "correo@gmail.co
```

YFphu0kLMna9K3mvqU7u	>	+ Añadir campo
ZqVp1P2p67KxXRnRszdM		correo: "correo@gmail.com"
manuel@gmail.com		usuario: "Pepe"

1. Añadiendo nosotros la clave, esta debe ser única para que el usuario se añada.

```
firebaseFirestore.collection("Usuarios").document(usuario.correo).set(usuario
```

maria@gmail.com	>	correo: "maria@gmail.com"
		usuario: "Maria"

Para insertar el valor del nuevo hijo hemos utilizado un objeto de la clase *Pojo Usuario*, que nos habremos creado con anterioridad, Firestore convierte automáticamente los atributos con sus valores para poder guardarlos correctamente. **Cuidado deberemos tener los atributos públicos o usar getter y setter.**

```
class Usuario : Serializable {
    lateinit var usuario: String
    lateinit var correo: String

    constructor() {}
    constructor(usuario: String, correo: String) {
        this.usuario = usuario
        this.correo = correo
    }
}
```

El código completo para crear la aplicación que nos añadirá un usuario cada vez que pulsemos el botón añadir, de forma que el id del usuario sea el correo, será el siguiente:

```

class MainActivity : AppCompatActivity() {
    lateinit var firebaseFirestore: FirebaseFirestore
    var listenerRegistration: ListenerRegistration? = null
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        firebaseFirestore = FirebaseFirestore.getInstance()
        val usuarioET = findViewById<TextInputLayout>(R.id.usuario)
        val correoET = findViewById<TextInputLayout>(R.id.correo)
        val añadir = findViewById<Button>(R.id.anyadir)

        val salida = findViewById<TextView>(R.id.salida)
        añadir.setOnClickListener{
            val usuario = Usuario(usuarioET.editText?.text.toString(),
                                   correoET.editText?.text.toString())
            firebaseFirestore.collection("Usuarios")
                               .document(usuario.correo)
                               .set(usuario)
                               .addOnSuccessListener {
19                               Toast.makeText(
                                   this,
                                   "Usuario Añadido",
                                   Toast.LENGTH_SHORT
                                   ).show()}
25                               .addOnFailureListener { e ->
                                   Toast.makeText(
                                   this,
                                   "Error" + e.message,
                                   Toast.LENGTH_SHORT
                                   ).show()}
            }
        }
    }
}

```

📌 como se puede ver en el código del OnClick, añadimos el usuario recogido de los EditText creando como clave el correo, como se ha explicado anteriormente en el tema. Además a los métodos de añadir objeto (set o add) se le pueden asignar distintos escuchadores para comprobar el estado del proceso. **Línea 19 y 25.**

Para [buscar un documento en Cloud Firestore](#), existen distintas posibilidades basadas en la cláusula **Where**. Por ejemplo, si quisiéramos comprobar si el id del usuario no está repetido y solo en ese caso añadirlo, podríamos modificar el anterior código de la siguiente manera:

```

fun compruebaSiExisteYAnade(usuario: Usuario) {
    firebaseFirestore.collection("Usuarios")
        .whereEqualTo(FieldPath.documentId(), usuario.correo).get()
        .addOnCompleteListener(OnCompleteListener<QuerySnapshot?> {
            task ->
                if (task.isSuccessful) {
                    if (task.result?.size() == 0) anyadeUsuario(usuario)
                    else Toast.makeText(
                        this,
                        "El correo ya existe, introduce uno nuevo",
                        Toast.LENGTH_LONG
                    ).show()
                } else {
                    Toast.makeText(this, task.exception.toString(),
                        Toast.LENGTH_LONG)
                        .show()
                }
            })
        })
}

fun anyadeUsuario(usuario: Usuario) {
    firebaseFirestore.collection("Usuarios")
        .document(usuario.correo).set(usuario)
        .addOnSuccessListener {
            Toast.makeText(
                this,
                "Usuario Añadido",
                Toast.LENGTH_SHORT
            ).show()
        }.addOnFailureListener { e ->
            Toast.makeText(
                this,
                "Error" + e.message,
                Toast.LENGTH_SHORT
            ).show()
        }
}

```

Si quisieramos dar funcionalidad al botón **Eliminar**, podemos hacer algo parecido a lo siguiente. En este caso se está eliminando por nombre de usuario, así que en el caso de existir más de un documento con el mismo nombre, se eliminarán todos.

```

fun Elimina(usuario: Usuario) {
    firebaseFirestore.collection("Usuarios")
        .whereEqualTo("usuario", usuario.usuario)
        .get()
        .addOnCompleteListener { task ->
            for (documento in task.result!!) documento.reference.delete()
        }
}

```


Si lo que queremos es controlar los cambios que ocurren en la BD, sea a través de una aplicación o directamente desde la consola de Firebase, tendremos que registrar un listener del tipo `ListenerRegistration`, que se inicializará sobre la consulta que deseemos con `addSnapshotListener`. En el siguiente ejemplo ponemos a escuchar todos los documentos de la colección `Usuarios`, mostrando en el `TextView` (que está bajo los botones) el resultado de cualquier modificación en cualquier documento de la colección.

```
fun listarUsuarios() {
    val query = firebaseFirestore.collection("Usuarios")
    listenerRegistration = query.addSnapshotListener {value, error->
        if (error == null) {

            for (dc in value!!.documentChanges) {
                when (dc.type) {
                    DocumentChange.Type.ADDED -> salida.text =
                        "${salida.text}\nSe ha añadido:" +
                        "${dc.document.data}\n".trimIndent()
                    DocumentChange.Type.MODIFIED -> salida.text =
                        "${salida.text}\n Se ha modificado:" +
                        "${dc.document.data}\n".trimIndent()
                    DocumentChange.Type.REMOVED -> salida.text =
                        "${salida.text}\nSe ha eliminado:" +
                        "${dc.document.data}\n".trimIndent()
                }
            }
        }
        else Toast.makeText(this, "No se puede listar"+error,
            Toast.LENGTH_SHORT).show()
    }
}
```

Se puede realizar la consulta sobre cualquier elemento de la colección después de haber sido seleccionado, de la siguiente manera:

```
query.whereEqualTo("correo", "manuel@gamil.com").addSnapshotListener{...}
```

 Otro tema importante a tener en cuenta, es que la suscripción a una referencia de una base de datos de Firebase, es decir, el hecho de asignar un listener a una ubicación del árbol para estar al tanto de sus cambios **no es algo gratuito** desde el punto de vista de consumo de recursos. Por tanto, es recomendable eliminar esa suscripción cuando ya no la necesitamos. Para hacer esto basta con llamar al método `remove()` del listener registrado, cuando no deseemos seguir escuchando.

```
override fun onDestroy() {  
    super.onDestroy()  
    listenerRegistration!!.remove()  
}
```

FirestoreUI y RecyclerView

FirestoreUI nos permite asociar fácilmente a un control **ListView** o **RecyclerView**, una referencia a una lista de elementos de una base de datos Firestore. De esta forma, el control se actualizará automáticamente cada vez que se produzca cualquier cambio en la lista, sin tener que gestionar manualmente por nuestra parte los eventos de la lista, ni tener que almacenar en una estructura paralela la información, ni tener que construir gran parte de los adaptadores necesarios... en resumen, ahorrando muchas líneas de código y evitando muchos posibles errores. Para utilizar **FirestoreUI** lo primero que tendremos que hacer será añadir la referencia a la librería en el fichero build.gradle de nuestro módulo principal. Hay que tener en cuenta que cada versión de **FirestoreUI** es compatible únicamente con una versión concreta de Firestore, por lo que debemos asegurar que las versiones utilizadas de ambas librerías son coherentes. En la página de **FirestoreUI** tenéis disponible la tabla de compatibilidades entre versiones. En el momento de actualización de estos apuntes:

```
implementation 'com.firebaseui:firebase-ui-firestore:8.0.0'
```

Esta librería nos provee de un Adaptador derivado de **RecyclerView.ViewHolder** que gestionará automáticamente la carga de los datos que le pasemos como referencia en la vista asignada por el **Holder**, para ello lo primero que deberemos crear es el ViewHolder personalizado que gestione los datos de nuestra aplicación (retomamos el ejemplo e implementamos el botón listar, podemos crear un fragment o activity para mostrar).

```

class Adaptador(options: FirestoreRecyclerOptions<Usuario>) :
2   FirestoreRecyclerAdapter<Usuario, Adaptador.Holder>(options),
   View.OnClickListener {
   private var listener: View.OnClickListener? = null
   override fun onBindViewHolder(holder: Holder, position: Int,
                                model: Usuario) {

       holder.bind(model)
   }
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
                                Holder {
       val view: View = LayoutInflater.from(parent.context)
           .inflate(R.layout.linea_recycler, parent, false)
       view.setOnClickListener(this)
       return Holder(view)
   }
   fun onClickListener(listener: View.OnClickListener?) {
       this.listener = listener
   }
   override fun onClick(v: View?) {
       listener?.onClick(v)
   }
22  inner class Holder(v: View) : RecyclerView.ViewHolder(v) {
       private val usuario: TextView
       private val correo: TextView
       fun bind(item: Usuario) {
           usuario.text = item.usuario
           correo.text = item.correo
28       }
       init {
           usuario = v.findViewById(R.id.usuario)
           correo = v.findViewById(R.id.correo)
       }
   }
}

```

📌 Definimos un adaptador como ya hemos hecho anteriormente, con el Holder para controlar las vistas de las líneas del recycler **Línea 22 a 28**, en donde sobrescribiríamos, obligatoriamente, los dos métodos y el constructor. He implementaríamos los listener necesarios para el funcionamiento de los eventos. La principal diferencia es que el Adaptador hereda de la clase

FirestoreRecyclerAdapter . FirebaseUI nos facilita el listado de los elementos extraídos de la BD proporcionando esta librería. A esta clase le tenemos que indicar que los elementos usados son: el ViewHolder personalizado que acabamos de crear y la clase java que utilizemos para encapsular la información de cada elemento de la lista

FirestoreRecyclerAdapter<Usuario, Adaptador.Holder> .

Al crear un objeto de esta clase, debemos pasarle un objeto de la misma librería, de tipo **FirestoreRecyclerOptions** .

```
val firestoreRecyclerOptions = FirestoreRecyclerOptions.Builder<Usuario>()
    .setQuery(query, Usuario::class.java).build()
```

Al que debemos pasarle la siguiente información:

- El objeto clase de nuestro Item (Usuario.class),
- La referencia al nodo de la base de datos que contiene la lista de elementos que queremos mostrar en el control.

Ahora solo quedaría crear un objeto del tipo adaptador que nos hemos creado, pasándole el elemento **firebaseRecyclerOption** y ya tendríamos casi todo hecho.

```
private fun cargarRecycler(query: Query) {
    val firestoreRecyclerOptions = FirestoreRecyclerOptions.
        Builder<Usuario>()
        .setQuery(query, Usuario::class.java).build()
    recyclerView = vista.findViewById(R.id.recycler)
    adapter = Adaptador(firestoreRecyclerOptions)
    //Click para eliminar elementos
    adapter!!.onClickListener{
        Toast.makeText(
            getActivity(),
            "Elemento eliminado" + recyclerView!!.
                getChildAdapterPosition(it),
            Toast.LENGTH_SHORT
        ).show()
        adapter!!.snapshots.getSnapshot(recyclerView!!.
            getChildAdapterPosition(it)).
            reference.delete()
    }
    recyclerView!!.adapter = adapter
    recyclerView!!.layoutManager = LinearLayoutManager(getActivity())
}
```

No deberemos olvidar iniciar el escuchador del adaptador al comenzar la aplicación y cerrarlo al acabar.

```
override fun onStart() {
    super.onStart()
    adapter!!.startListening()
}
```

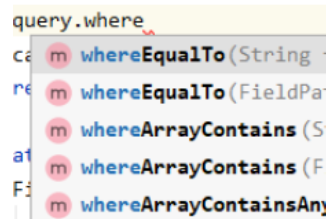
```
override fun onStop() {
    super.onStop()
    adapter!!.stopListening()
}
```


Filtrado y ordenación

Lo primero que debemos tener en cuenta es que en las BD de Firebase no vamos a tener todas las facilidades de ordenación y filtrado que suelen encontrarse en bases de datos SQL tradicionales. Para las consultas podemos encontrar la información que necesitamos en el siguiente enlace

<https://firebase.google.com/docs/firestore/query-data/queries>.

Para realizar consultas se utiliza cualquiera de los métodos `where` que indican que tipo de consulta necesitamos:



Usando la colección del ejemplo que nos encontramos en el enlace de la página de documentación de firestore, *con la expresión siguiente recuperaríamos todos los documentos que tienen una población menor a 100000 habitantes.*

```
citiesRef.whereLessThan("population", 100000);
```

Se pueden enlazar búsquedas con `where`, de la siguiente manera:

```
citiesRef.whereEqualTo("state", "CA").whereLessThan("population", 100000);
```

En este caso se buscarán todos los documentos de estado igual a CA y del resultado de esa búsqueda se extraerán solo aquellos que tengan una población menor a 100000.

También está la opción de buscar dentro de una colección directamente. Si algunos de nuestros documentos tienen un miembro que es una colección, podremos realizar la consulta sobre este elemento usando alguna de las sobrecargas de

`whereArrayContains`

```
citiesRef.whereArrayContains("regions", "west_coast");
```

En los datos 'regions' se refiere a un array con los nombres de las regiones a las que pertenece cada ciudad, por lo que con la consulta anterior se extraerán todas las ciudades que en ese array tengan la entrada 'west_coast', si la tienen repetida solo aparecerán una vez en los resultados.

Si lo que queremos hacer es ordenar los datos obtenidos, tenemos toda la información en la url <https://firebase.google.com/docs/firestore/query-data/orderlimitdata?hl=es> Se utilizará cualquiera de las sobrecargas del método

`orderBy`

```

query.or
: m orderBy(String field)
: m orderBy(FieldPath fieldPath)
: m orderBy(String field, Direction directi
it m orderBy(FieldPath fieldPath, Direction

```

Por ejemplo, se pueden combinar más de un **orderBy** para realizar la consulta que necesitamos. En este caso se ordena por el nombre del estado y a partir de ahí por número de población y en orden descendiente.

```
citiesRef.orderBy("state").orderBy("population", Direction.DESENDING);
```

Una cláusula **orderBy** también filtra en busca de la existencia del campo dado. El conjunto de resultados no incluirá documentos que no contengan el campo correspondiente. Como es de esperar, también se pueden ordenar los datos después de realizar una consulta **where**, pero con la condición que debe ser sobre el mismo campo por el que se ha hecho la búsqueda.

```
citiesRef.whereGreaterThan("population", 100000).orderBy("population");
```

Si con **orderBy** se puede especificar el orden de clasificación de los datos, con **limit** puedes limitar la cantidad de documentos recuperados.

Devuelve los nombres de las tres primeras ciudades

```
citiesRef.orderBy("name").limit(3);
```

Devuelve los nombres de las tres últimas ciudades

```
citiesRef.orderBy("name", Direction.DESENDING).limit(3);
```

Otros cursores de consultas que están disponibles son:

- **startAt** -> La consulta sólo devolverá los elementos cuyo valor sea igual o superior al dato pasado como parámetro.

Ciudades con número de habitantes 100000 y ordenados por número de población

```
db.collection("cities")
  .orderBy("population")
  .startAt(100000);
```

- **startAfter** -> Idéntico al anterior pero para valores superiores, no incluye los documentos con valor igual.

- **endAt** -> La consulta sólo devolverá los elementos cuyo valor sea igual o inferior al dato pasado como parámetro.

```
db.collection("cities")
  .orderBy("population")
  .endAt(100000);
```

*Ciudades con número de habitantes <= 100000
y ordenados por número de población*

- **endBefore** -> Idéntico al anterior pero para valores inferiores, no incluye los documentos con valor igual.

También podremos utilizar varios criterios de filtrado en la misma consulta, es decir podremos combinar varios de los métodos anteriores para obtener sólo el rango de elementos necesario.

```
Query next = db.collection("cities")
    .orderBy("population")
    .startAfter(lastVisible)
    .limit(25);
```

Firestore Auth. Autenticación

Firestore Authentication proporciona servicios de backend, SDK fáciles de usar y bibliotecas de IU ya elaboradas para autenticar a los usuarios en la app. *Vamos a iniciarnos en su uso modificando el proyecto anterior, para ello crearemos otro Fragment que será el primero que se inicie en la aplicación. Nos permitirá crear una cuenta o autenticarnos con una ya existente, si la autenticación es correcta se cargará el fragment con el que se iniciaba nuestro proyecto anteriormente.*

Si queremos autenticarnos como usuarios, tenemos varias

opciones: **email/password, google, teléfono, Facebook, twitter, github, anonymous** .

* Nosotros vamos a ver el ejemplo de email/password y el de anonymous, en el resto de autenticaciones podéis seguir la documentación de FireBase.*

El primer paso sería añadir la dependencia de autenticación, que como tenemos incluido la de firebase-bom, no necesitará número de versión:

```
implementation 'com.google.firebase:firebase-auth'
```

Tendremos que codificar un layout que permita introducir el usuario y la contraseña para poder logearnos.

Usuario

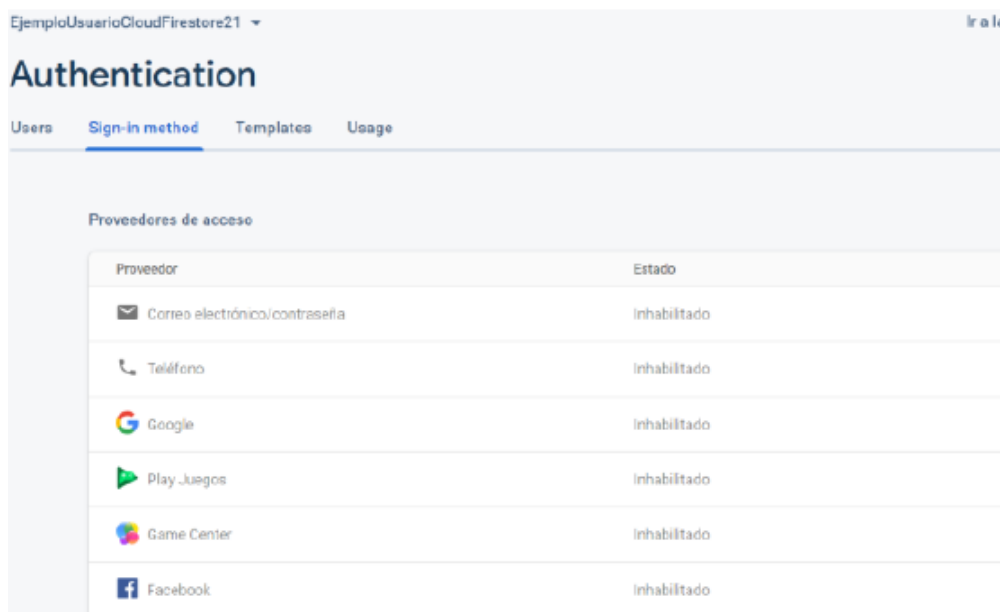
Contraseña

SIGNIN

SIGNUP

ANONIMO

Lo siguiente será activar los proveedores con los que queremos iniciar sesión. Nos vamos a la opción Authentication y dentro de este a la pestaña Métodos de inicio de sesión. Tendremos que seleccionar ambos casos y habilitar el estado.



Una vez añadido los proveedores, tendremos dos formas de crear los usuarios: **a través de la consola y desde la aplicación**. Desde la consola nos tendremos que ir a la pestaña usuarios, aquí podremos crear los usuarios que necesitemos. Los usuarios anónimos aparecerán solamente con el id único que asigna Firebase a cada usuario creado.

El otro caso será al codificar la aplicación. Para todos los casos de autenticación necesitamos añadir un escuchador a una variable de tipo

`FirebaseAuth.AuthStateListener`. Este escuchador será asignado en el método `onStart` y eliminado en `onStop`, y su función es la de controlar cualquier cambio que ocurra con el usuario.

```


class FragmentLogearse : Fragment() {
    lateinit var mAuthListener: AuthStateListener
    lateinit var mAuth: FirebaseAuth
    lateinit var user: TextInputLayout
    lateinit var password: TextInputLayout
    lateinit var interfaceFragments: InterfaceFragments
    override fun onStart() {
        super.onStart()
        mAuth.addAuthStateListener(mAuthListener)
    }
    override fun onStop() {
        super.onStop()
        mAuth.removeAuthStateListener(mAuthListener)
        mAuth.signOut()
    }
    override fun onAttach(context: Context) {
        super.onAttach(context)
        interfaceFragments = context as InterfaceFragments
    }
}

```

```

////Escuchador de cambios en los usuarios
mAuthListener = AuthStateListener
{
    firebaseAuth ->
    val user = firebaseAuth.currentUser
    if (user != null) {
        Toast.makeText(getActivity(), user.email + " LOGEADO",
            Toast.LENGTH_SHORT).show()
        mAuth = firebaseAuth
    }
    else Toast.makeText(getActivity(), "Usuario NULO",
        Toast.LENGTH_SHORT).show()
}

```

 El escuchador AuthStateListener deberá ser incluido en el método onCreate de la aplicación.

El siguiente paso será decidir si queremos permitir crear usuarios nuevos, o solamente logearnos con alguno existente. En nuestra aplicación vamos a hacer las dos cosas, e incluso nos logearemos como anónimos. El código aparece a continuación, y un enlace con toda la información

<https://firebase.google.com/docs/auth/android/password-auth>

```

//////// Crear usuario nuevo y iniciar sesión
btCrear.setOnClickListener{
    mAuth.createUserWithEmailAndPassword(
        user.editText!!.text.toString(), password.editText!!
            .text.toString()
    )
    .addOnCompleteListener(requireActivity(),
        OnCompleteListener<AuthResult?> { task ->
            if (task.isSuccessful) {
                Toast.makeText(getActivity(),"Usuario creado",
                    Toast.LENGTH_SHORT).show()
                iniciarFragmen(task.result?.getUser()?.
                    getEmail()?.split("@")!![0])
            } else Toast.makeText(
                getActivity(),
                "Problemas al crear usuario" +task.exception,
                Toast.LENGTH_SHORT
            ).show()
        })
    })
}
return view

```

```

////////Iniciar sesión con usuario y contraseña
btIniciar.setOnClickListener {
    mAuth.signInWithEmailAndPassword(
        user.editText!!.text.toString(), password.editText!!
            .text.toString()
    )
    .addOnCompleteListener(requireActivity(),
        OnCompleteListener<AuthResult?> { task ->
            if (!task.isSuccessful) {
                Toast.makeText(
                    getActivity(),
                    "Authentication failed:" + task.exception,
                    Toast.LENGTH_SHORT
                ).show()
            } else iniciarFragmen(task.result?.getUser()?.
                getEmail()?.split("@")!![0])
        })
    })
}

```

```

////////// Iniciar sesión con anónimo
btAnonimus.setOnClickListener{
    mAuth.signInAnonymously().addOnCompleteListener
    (
        requireActivity(),
        OnCompleteListener<AuthResult?> { task ->
            if (!task.isSuccessful) {
                Toast.makeText(
                    getActivity(),
                    "Authentication failed:" + task.exception,
                    Toast.LENGTH_SHORT
                ).show()
            } else iniciarFragmen("anonymous")
        }
    )
}
}

```

Como podemos ver en el código, lo que se hace es añadir el mismo escuchador a cualquiera de los métodos que hayamos elegido (login, anónimo o añadir usuario) sobre un objeto **FirebaseAuth** . Con estos pasos tendremos la autenticación controlada, y podremos seguir con nuestra aplicación.

 **Resuelve el ejercicio de los ejemplos hasta conseguir un correcto funcionamiento**