

Apuntes

[Descargar estos apuntes](#)

Resumen B11. 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. [Acceso a bases de datos con ApiRest y Retrofit](#)
 1. [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

SQLite

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`. 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

```

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)
    }
}

```

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

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.

```

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()
    }
}

```



Se ha creado nuestra base de datos 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

Acceso a la BD mediante SQLiteDatabase

La otra forma de acceder a la BD es utilizar los métodos específicos **insert()**, **update()** y **delete()** de la clase **SQLiteDatabase**.

- **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" 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.

Recuperación de datos con rawQuery

```
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
}
```

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")

```

```

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
    }
}

```

Uso de SimpleCursorAdapter


```

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)
}
```

Cursores con RecyclerView

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)
    }
    abstract fun onBindViewHolder(holder: RecyclerView.ViewHolder,
                                cursor: Cursor)
    override fun getItemCount(): Int
    {
        return if (mCursor != null)    mCursor.getCount()
            else 0
    }
    init {
        mCursor = cursor
    }
}
```

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
        }
    }
}

```

3. 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)
    }
}
```

Acceso a bases de datos remotas

Acceso a bases de datos con ApiRest y Retrofit

Consumo de un Servicio Rest desde Android

Retrofit la utilizaremos para hacer peticiones y procesar las respuestas del API Rest, 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.

```
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)
}
```

2. Creación de las clases Pojo que le servirá al Gson para parsear los resultados.

```
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. 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>  
}
```

[<http://square.github.io/retrofit/>]

💡 **Retrofit** para invocar la url de la petición usa **@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 vayan 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.

Si no vamos a esperar a que acabe la corrutina:

```
private fun anyadirUsuario(usuario: Usuario) {
    val context=this
    var salida:String?
    val proveedorServicios: ProveedorServicio = crearRetrofit()
    CoroutineScope(Dispatchers.IO).launch {
        val response = proveedorServicios.insertarUsuario(usuario)
        if (response.isSuccessful) {
            val usuariosResponse = response.body()

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

Y este sería un ejemplo, en el que esperamos a que termine.

Si en el APIAdapter me creo una función cómo esta, que implementa la interfaz

Deferred [<https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/index.html>] , y lanza una corrutina de **modo asíncrono** [<https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html>]:

```

fun seleccionarFotos(context: Context): Deferred<ArrayList<Dato>> {
    val proveedorServicios: ProveedorServicios = crearRetrofit()
    return CoroutineScope(Dispatchers.IO).async {
        var datosDev=ArrayList<Dato>()
        val response: Response<ArrayList<Dato>>
        response=proveedorServicios.getFotos()
        if (response.isSuccessful) {
            datosDev = response.body()!!
        }
        datosDev
    }
}

```

Posteriormente podré llamarla diciendo que espere a que dicha función termine mediante **.await()** al final:

```

MainActivity.datos =
    ApiRestAdapter.seleccionarFotos(requireContext()).await()

```

🚩 Hay que tener en cuenta que si dentro de una corrutina necesito acceder a this, debo grabarlo anteriormente en una variable, ya que dentro de la rutina this me dará error.

```

fun prepararDatos(tipo:Int, texto:String)
{
    val contextoAplicacion:Context=this
    CoroutineScope(Dispatchers.IO).launch >{

        val adaptadorAPI=AdaptadorAPI>(contextoAplicacion)
        ...
    }
}

```


Acceso a Base de Datos con FIREBASE

Firebase

Creando un app de Firebase


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'  
    ...  
}
```

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

Firestore DataBase

[Firebase Realtime Database](#)

[Firebase Firestore DataBase](#)

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 >

ZqVp1P2p67KxXRnRsZdM

manuel@gmail.com

+ Añadir campo

correo: "correo@gmail.com"

usuario: "Pepe"

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

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

maria@gmail.com >

correo: "maria@gmail.com"

usuario: "Maria"

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 {
                                   Toast.makeText(
                                       this,
                                       "Usuario Añadido",
                                       Toast.LENGTH_SHORT
                                   ).show()}
                               .addOnFailureListener { e ->
                                   Toast.makeText(
                                       this,
                                       "Error" + e.message,
                                       Toast.LENGTH_SHORT
                                   ).show()}
        }
    }
}

```

[Buscar un documento en Cloud Firestore](#), existen distintas posibilidades basadas en la clausula **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()
}

```

FirebaseUI y RecyclerView

FirebaseUI

En el momento de actualización de estos apuntes:

```

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

```

```

class Adaptador(options: FirestoreRecyclerOptions<Usuario>) :
    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)
    }
    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
        }
        init {
            usuario = v.findViewById(R.id.usuario)
            correo = v.findViewById(R.id.correo)
        }
    }
}

```

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()
```

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.

Filtrado y ordenación

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

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

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

```
citiesRef.whereEqualTo("state", "CA").whereLessThan("population", 1000000);
```

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");
```

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

```
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.

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

Otros cursores de consultas que están disponibles son: **startAt**, **startAfter**, **endAt**, **endBefore**

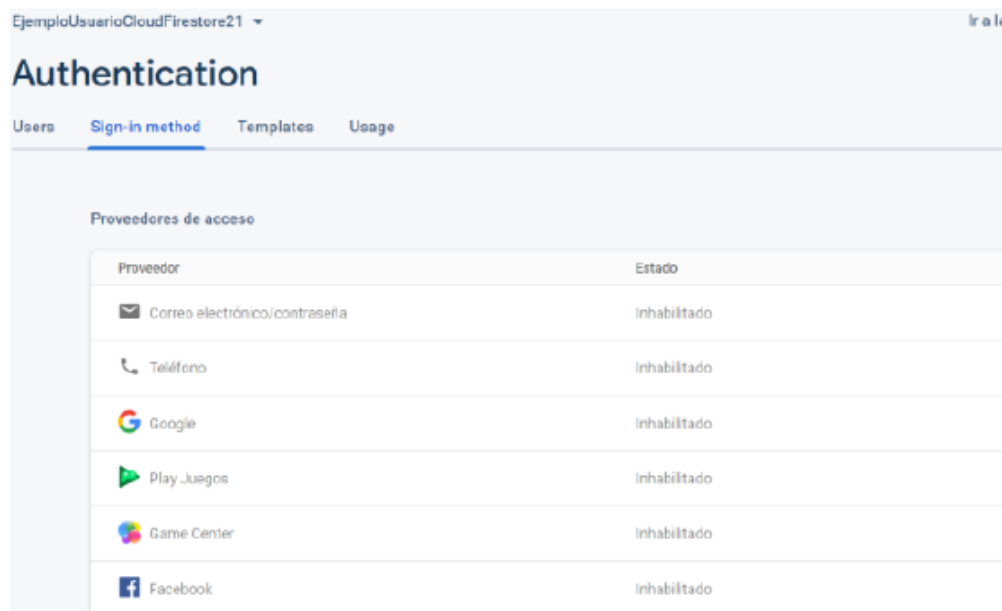
```
Query next = db.collection("cities")
    .orderBy("population")
    .startAfter(lastVisible)
    .limit(25);
```

Firestore Auth. Autenticación

Firestore Authentication

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

Después de tener el layout para identificarse, 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.



Tendremos dos formas de crear los usuarios: **a través de la consola y desde la aplicación.**

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

```

//////// Crear usuario nuevo y iniciar sesión
btCrear.setOnClickListener{
    FirebaseAuth.getInstance().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 {
        FirebaseAuth.getInstance().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{
        FirebaseAuth.getInstance().signInAnonymously().addOnCompleteLis
        (
            requireActivity(),
            OnCompleteListener<AuthResult?> { task ->
                if (!task.isSuccessful) {
                    Toast.makeText(
                        getActivity(),
                        "Authentication failed:" + task.exception,
                        Toast.LENGTH_SHORT
                    ).show()
                } else iniciarFragmen("anonymous")
            }
        )
    }
}

```