



PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TEMA 10:
ALMACENAMIENTO
PERSISTENTE DE DATOS

ÍNDICE

1. La clase Shared Preferences.
2. Archivos de Texto.
3. SQLite
4. Conexión Firebase



PMDM

ALMACENAMIENTO PERSISTENTE DE
DATOS



1. Clase SharedPreferences

1.- Clase SharedPreferences

- Android nos ofrece 3 alternativas para el almacenamiento permanente de datos (es decir, que estos no se borran cuando se cierra la aplicación o apagamos el móvil).
- Según el tipo de necesidades utilizaremos alguno de estos métodos:
 - Mediante la clase SharedPreferences (archivo XML).
 - Mediante archivos de Texto.
 - En una base de datos con acceso a SQL.
- Además, perfectamente una aplicación puede utilizar más de uno de estos métodos para el almacenamiento de datos.

1.- Clase SharedPreferences

❖ La clase SharedPreferences:

- Cuando tenemos que almacenar una pequeña cantidad limitada de datos es adecuado utilizar la clase SharedPreferences. Por ejemplo, configuraciones de la aplicación como pueden ser colores de pantalla, nivel actual en un juego, opciones elegidas en una aplicación...
- Para probar el funcionamiento de este tipo de almacenamiento vamos a realizar la siguiente aplicación:

1.- Clase SharedPreferences

- Actividad que solicite el email de una persona. Guardar el email ingresado utilizando la clase SharedPreferences. Cada vez que se inicie la aplicación, cargar en el EditText el último email ingresado. Disponer un botón para almacenar el email ingresado y finalizar el programa.
- Quedaría de la siguiente manera:



1.- Clase SharedPreferences

- **PARA GUARDAR EL DATO**, lo primero que se hace es crear un objeto de la clase SharedPreferences:

val preferencias: SharedPreferences = getSharedPreferences ("datos", Context.MODE_PRIVATE)

- “datos” será el nombre del archivo, y la forma de creación del archivo “MODE_PRIVATE” indica que solo esta aplicación puede consultar el archivo XML que se crea.
- Si necesitas que tu app comparta este archivo con otras apps deberás utilizar el componente *FileProvider*. Para más información, visita:

<https://developer.android.com/training/secure-file-sharing?hl=es-419>

1.- Clase SharedPreferences

- Después creamos un objeto de la clase Editor y obtenemos la referencia del objeto de la clase SharedPreferences que acabamos de crear. Mediante el método putString almacenamos en la variable “email” el valor del String cargado en el EditText:

```
val editor: SharedPreferences.Editor = preferencias.edit()  
editor.putString("email", editTextEMail.text.toString())
```

- Cuando guardamos datos en el archivo de preferencias, podemos almacenar distintos tipos de datos según el método que llamemos en el momento de grabar:
 - editor.putInt("edad",3);
 - editor.putBoolean("activo", true);
 - editor.putFloat("altura", 2.3f);

1.- Clase SharedPreferences

- Por último debemos llamar al método commit de la clase Editor para que el dato quede almacenado de forma permanente en el archivo de preferencias. Esto hace que cuando volvamos a arrancar la aplicación se recupere el último email ingresado.

editor.commit();

1.- Clase SharedPreferences

- **PARA RECUPERAR EL DATO**, y que aparezca en el EditText cuando abramos nuestra aplicación, crearemos en el método OnCreate un objeto de la clase SharedPreferences, recuperando a continuación el dato y asignándoselo al EditText:

```
val preferencias: SharedPreferences=getSharedPreferences("datos", Context.MODE_PRIVATE)  
  
val email: String? = preferencias.getString("email", "")  
  
if(email != null) {  
  
    editTextEmail.setText(email)  
  
}
```

- Para extraer los datos del archivo de preferencias debemos indicar el nombre a extraer y un valor de retorno si dicho nombre no existe en el archivo de preferencias (en nuestro ejemplo, la primera vez que se ejecute nuestro programa, como es lógico, no existe el archivo de preferencias, lo que hace que Android lo cree. Si tratamos de extraer el valor de email retornará el segundo parámetro (es decir, el String con una cadena vacía "")).

EJERCICIOS

- **Ejercicio 01.- (OPTATIVO):** Crea un proyecto en Android Studio llamado “Mini Agenda” que contenga una Actividad.
- Este proyecto consistirá en crear una pequeña agenda mediante la clase SharedPreferences.

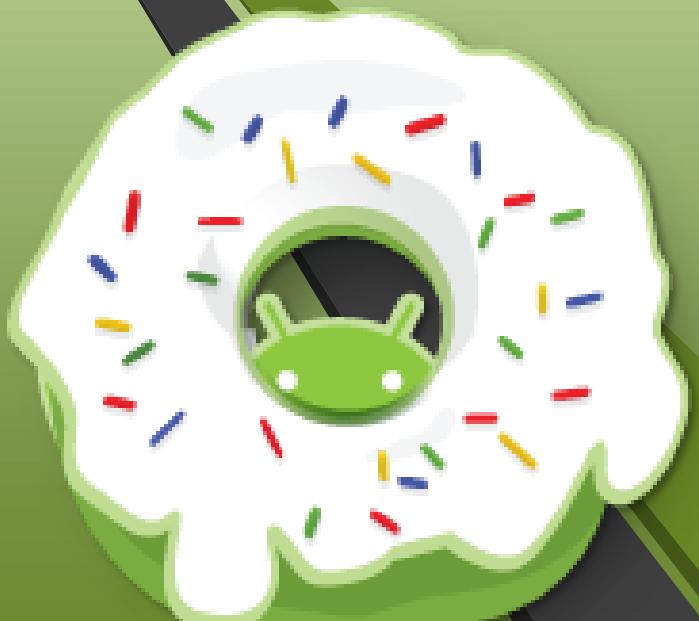
EJERCICIOS

- Quedaría de la siguiente manera:



PMDM

ALMACENAMIENTO PERSISTENTE DE DATOS



2. Archivos de Texto

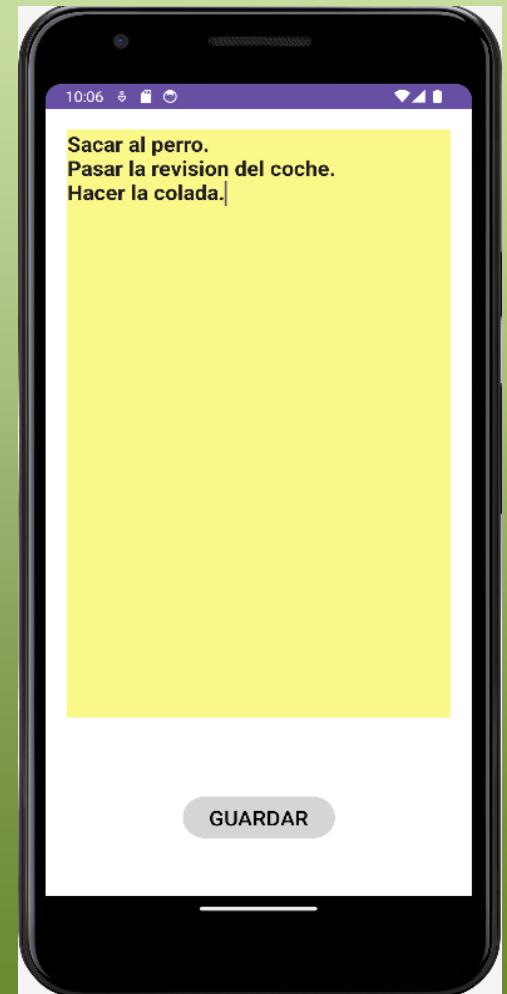
2.- Archivos de Texto

❖ Archivos de texto:

- Otra posibilidad de almacenar datos en nuestro dispositivo Android es el empleo de un archivo de texto que se guardará en el almacenamiento interno del dispositivo (la otra posibilidad sería almacenarlo en una tarjeta SD)
- Para probar el funcionamiento de este tipo de almacenamiento vamos a realizar la siguiente aplicación:

2.- Archivos de Texto

- Actividad donde podemos escribir las tareas cotidianas para que no se nos olviden. Al salir de la aplicación se guardaran en un fichero de texto para que, al volver a abrir la aplicación, nos aparezcan las que habíamos escrito:



2.- Archivos de Texto

- Para leer y escribir ficheros en Android con lenguaje KOTLIN, se hace de la misma manera que en JAVA, ya que necesitaremos dos flujos para la lectura y uno solo para la escritura.
- Además, también irán entre los bloques try-catch, y deberemos cerrar los flujos después de utilizarlos con el método close().

2.- Archivos de Texto

- Para la lectura necesitaremos las clases *InputStreamReader* y *BufferedReader*.
- Primero creamos un objeto de la clase *InputStreamReader*, pasándole un objeto *FileInputStream* que obtenemos con el método *openFileInput*, el cual tiene como parámetro el nombre del archivo que queremos leer.
- Luego, crearemos el flujo *BufferedReader* al que le pasaremos el flujo anterior en el constructor, y utilizaremos su método *readLine()* para leer el fichero (línea a línea).

- Por tanto, con el método `readLine()` de la clase `BufferedReader`, iremos leyendo línea a línea:

2.- Archivos de Texto

```
try {  
    val fileInputStream = applicationContext.openFileInput( name: "tareas.txt")  
    val inputStreamReader = InputStreamReader(fileInputStream)  
    val bufferedReader = BufferedReader(inputStreamReader)  
  
    val tareasBuilder: StringBuilder = StringBuilder()  
  
    var linea: String? = bufferedReader.readLine()  
    while(linea != null) {  
        tareasBuilder.append(linea)  
        tareasBuilder.append("\n")  
        linea = bufferedReader.readLine()  
    }  
  
    bufferedReader.close()  
    inputStreamReader.close()  
    fileInputStream.close()  
  
    editTextTareas.setText(tareasBuilder.toString())  
}  
} catch (e: IOException) {  
    e.printStackTrace()  
}
```

2.- Archivos de Texto

- Para escribir solo necesitaremos la clase *OutputStreamWriter*.
- Primero creamos un objeto de la clase *OutputStreamWriter* pasándole un objeto *FileOutputStream* que obtenemos con el método *openFileOutput*, que tiene dos parámetros: el nombre del archivo y el modo de acceso (*Context.MODE_PRIVATE*).

2.- Archivos de Texto

- Luego, bastará con llamar al método write al que le pasaremos el texto a escribir en el fichero.

```
try {
    val fileOutputStream = applicationContext.openFileOutput( name: "tareas.txt", Context.MODE_PRIVATE)
    val outputStreamWriter = OutputStreamWriter(fileOutputStream)
    outputStreamWriter.write(editTextTareas.text.toString())

    outputStreamWriter.close()
    fileOutputStream.close()

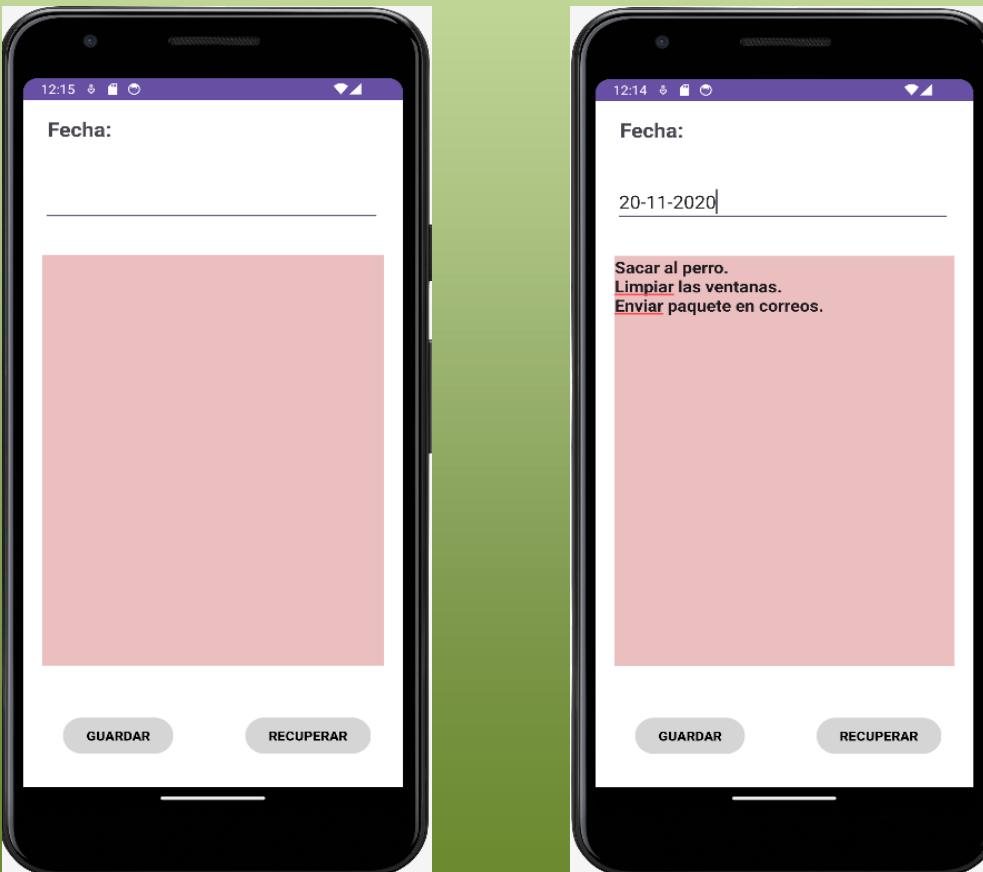
    Toast.makeText(applicationContext, R.string.tareas_guardadas_ok, Toast.LENGTH_SHORT).show()
} catch (e: FileNotFoundException) {
    e.printStackTrace()
} catch (e: IOException) {
    e.printStackTrace()
}
```

EJERCICIOS

- **Ejercicio 02.- (OPTATIVO):** Crea un proyecto en Android Studio llamado “Mis Tareas” que contenga una Actividad.
- Este proyecto consistirá en crear pequeños archivos de texto uno por cada fecha que queramos guardar. Luego, poniendo una fecha, podremos recuperar esas tareas.

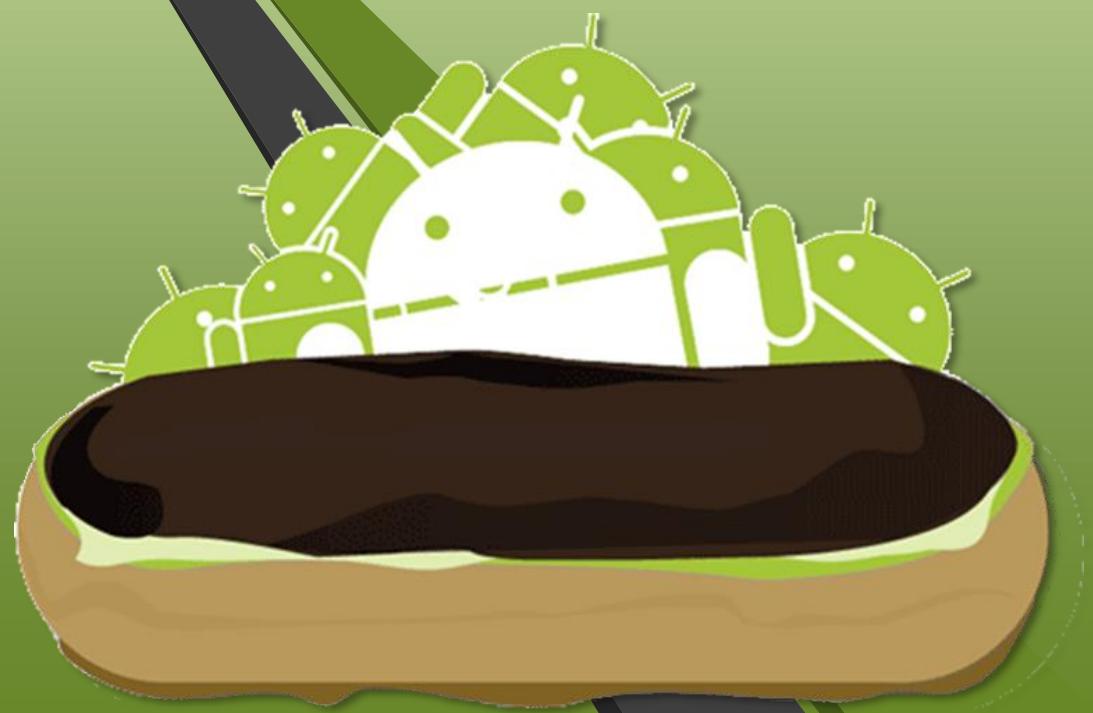
EJERCICIOS

- Quedaría de la siguiente manera:



PMDM

ALMACENAMIENTO PERSISTENTE DE
DATOS



3. SQLite

3.- SQLite

❖ SQLite:

- Hemos visto hasta ahora dos modos de almacenar datos de forma permanente (archivos de texto y la clase SharedPreferences).
- Otra herramienta nativa de Android para almacenar datos es en una base de datos llamada SQLite, que es una base de datos Open Source.
- Las ventajas que presenta utilizar SQLite es que no requiere configuración ni tiene un servidor de base de datos ejecutándose en un proceso separado. Además es relativamente simple su empleo.

3.- SQLite

- Para ver el funcionamiento de SQLite vamos a crear una pequeña base de datos de una tienda que guarde su stock de artículos en una tabla, que contendrá los siguientes campos: código, descripción, stock y precio.
- El programa debe permitir:
 1. Añadir artículos.
 2. Consulta de artículos por el código.
 3. Consulta de artículos por la descripción.
 4. Borrado de un artículo ingresando su código.
 5. Modificación de la descripción, stock y el precio.

3.- SQLite

- Lo primero que haremos es crear una nueva clase. Esta va a heredar de **SQLiteOpenHelper** y nos permitirá crear la base de datos, así como actualizar la estructura de tablas y datos iniciales. La vamos a llamar *AdministracionSQLiteOpenHelper*.

3.- SQLite

The screenshot shows the Android Studio interface. On the left is the Project Navigational Drawer, which lists the project structure under 'app': 'manifests', 'java' (containing 'com.example.sqlite' with files 'AdministracionSQLiteOpenHelper' and 'MainActivity'), 'res' (containing 'drawable', 'layout', 'mipmap', and 'values' with files 'colors.xml', 'dimens.xml', 'strings.xml', 'themes.xml', and 'themes.xml (night)'), and 'xml'. The 'AdministracionSQLiteOpenHelper' file is currently selected and shown in the main code editor. The code defines a class 'AdministracionSQLiteOpenHelper' that extends 'SQLiteOpenHelper'. It contains two overridden methods: 'onCreate' and 'onUpgrade', both of which are marked with a TODO comment: 'TODO(reason: "Not yet implemented")'. The code editor has syntax highlighting for Java and Kotlin.

```
1 package com.example.sqlite
2
3 import android.content.Context
4 import android.database.sqlite.SQLiteDatabase
5 import android.database.sqlite.SQLiteOpenHelper
6
7 class AdministracionSQLiteOpenHelper(context: Context, name: String, version: Int)
8     : SQLiteOpenHelper(context, name, null, version) {
9
10    override fun onCreate(db: SQLiteDatabase) {
11        TODO("Not yet implemented")
12    }
13
14    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
15        TODO("Not yet implemented")
16    }
17
18}
```

3.- SQLite

- Ahora, en el método **onCreate** de la clase que hemos elaborado, creamos la tabla de *artículos* de la tienda, con los campos: *código* de tipo entero y clave primaria, *descripción* de tipo texto, *stock* de tipo entero y *precio* de tipo real:

```
override fun onCreate(db: SQLiteDatabase) {
    db.execSQL(
        sql: "CREATE TABLE articulos (codigo INTEGER PRIMARY KEY, descripcion TEXT, stock INTEGER, precio REAL)"
    )
}
```

- El método **onCreate** se ejecuta una única vez, por lo que si, en un futuro quisiéramos modificar la estructura de la tabla, deberíamos hacerlo en el método **onUpgrade**.

3.- SQLite

- Lo siguiente será crear una sencilla interfaz:



3.- SQLite

- Para programar el botón de ALTA crearemos 3 objetos:
- Un objeto “admin” de la clase que creamos en el primer paso, **AdministracionSQLiteOpenHelper**, al que le pasamos this o applicationContext (referencia del Activity actual), “tienda” (es el nombre de la base de datos que crearemos en caso de que no exista) y un 1 indicando que es la primera versión de la base de datos (en caso que cambiemos la estructura o agreguemos tablas, por ejemplo, podemos pasar un 2 en lugar de un 1 para que se ejecute el método onUpgrade, donde indicaremos la nueva estructura de la base de datos).

3.- SQLite

- Un objeto “bd” de la clase **SQLiteDatabase**, llamando al método `getWritableDatabase` (`writableDatabase`) del objeto “admin” (la base de datos se abre en modo lectura y escritura).
- Un objeto “registro” de la clase **ContentValues** en el que, mediante el método `put`, inicializamos todos los campos a cargar.
- Ya solo nos queda llamar al método **insert** del objeto “bd” pasando en el primer parámetro el nombre de la tabla, como segundo parámetro un null y por último el objeto de la clase `ContentValues` ya inicializado.
- Cerramos la base de datos (`bd.close()`), limpiamos los `EditText` y mostramos un mensaje al usuario informándole de que los datos se han dado de alta correctamente:

3.- SQLite

```
private fun alta() {
    val codigo: Int? = editTextCodigo.text.toString().toIntOrNull()
    val descripcion: String = editTextDescripcion.text.toString()
    val stock: Int? = editTextStock.text.toString().toIntOrNull()
    val precio: Float? = editTextPrecio.text.toString().toFloatOrNull()

    if(codigo == null || descripcion == "" || stock == null || precio == null) {
        Toast.makeText(applicationContext, R.string.no_datos_articulo_msg, Toast.LENGTH_SHORT).show()
    } else {
        val admin: AdministracionSQLiteOpenHelper =
            AdministracionSQLiteOpenHelper(applicationContext, name: "tienda", version: 1)
        val bd: SQLiteDatabase = admin.writableDatabase
        val registro: ContentValues = ContentValues()
        registro.put("codigo", codigo)
        registro.put("descripcion", descripcion)
        registro.put("stock", stock)
        registro.put("precio", precio)
        bd.insert( table: "articulos", nullColumnHack: null, registro)
        bd.close()
        //Se limpian los EditText
        editTextCodigo.setText("")
        editTextDescripcion.setText("")
        editTextStock.setText("")
        editTextPrecio.setText("")
        //Se muestra mensaje de datos guardados correctamente
        Toast.makeText(applicationContext, R.string.articulo_guardado_ok_msg, Toast.LENGTH_SHORT).show()
    }
}
```

3.- SQLite

- Para programar el botón de BAJA es similar al de alta, solo que ahora en vez de `bd.insert()` utilizaremos el método **delete** de la clase `SQLiteDatabase`, al que le pasamos como primer parámetro el nombre de la tabla, en el segundo la condición del `where` ("`codigo=$código`") para filtrar la fila que borraremos de la tabla y como último parámetro `null` (correspondiente a los atributos del `where`, que no asignamos).
- Además, el método `delete` retorna un entero que indica la cantidad de registros borrados.

3.- SQLite

```
private fun bajaPorCodigo() {
    val codigo: Int? = editTextCodigo.text.toString().toIntOrNull()
    if(codigo == null) {
        Toast.makeText(applicationContext, R.string.no_codigo_articulo_msq, Toast.LENGTH_SHORT).show()
    } else {
        val admin: AdministracionSQLiteOpenHelper =
            AdministracionSQLiteOpenHelper(applicationContext, name: "tienda", version: 1)
        val bd: SQLiteDatabase = admin.writableDatabase
        val registrosBorrados: Int =
            bd.delete( table: "articulos", whereClause: "codigo=$codigo", whereArgs: null)
        if(registrosBorrados == 1) {
            Toast.makeText(applicationContext, R.string.articulo_borrado_ok_msq, Toast.LENGTH_SHORT).show()
            //Se limpian los EditText
            editTextCodigo.setText("")
            editTextDescripcion.setText("")
            editTextStock.setText("")
            editTextPrecio.setText("")
        } else {
            Toast.makeText(applicationContext, R.string.no_articulo_cod_msq, Toast.LENGTH_SHORT).show()
        }
    }
}
```

3.- SQLite

- La programación del botón de MODIFICACIÓN sería similar al de alta, pero debemos crear un objeto de la clase ContentValues en el que, mediante el método put, almacenemos los valores para cada campo que será modificado.
- Luego llamamos al método **update** de la clase SQLiteDatabase pasando el nombre de la tabla, el objeto de la clase ContentValues, la condición del where ("codigo=\$código") y, por último, un null (correspondiente a los atributos del where, que no asignamos).
- Además, el método update retorna un entero que indica la cantidad de registros actualizados.

3.- SQLite

```
private fun modificacionPorCodigo() {
    val codigo: Int? = editTextCodigo.text.toString().toIntOrNull()
    val descripcion: String = editTextDescripcion.text.toString()
    val stock: Int? = editTextStock.text.toString().toIntOrNull()
    val precio: Float? = editTextPrecio.text.toString().toFloatOrNull()

    val admin: AdministracionSQLiteOpenHelper =
        AdministracionSQLiteOpenHelper(applicationContext, name: "tienda", version: 1)
    val bd: SQLiteDatabase = admin.writableDatabase
    val registro: ContentValues = ContentValues()
    registro.put("codigo", codigo)
    registro.put("descripcion", descripcion)
    registro.put("stock", stock)
    registro.put("precio", precio)
    val registrosModificados: Int =
        bd.update( table: "articulos", registro, whereClause: "codigo=$codigo", whereArgs: null)
    bd.close()

    if(registrosModificados == 1) {
        Toast.makeText(applicationContext, R.string.articulo_modificado_ok_msg, Toast.LENGTH_SHORT).show()
        //Se limpian los EditText
        editTextCodigo.setText("")
        editTextDescripcion.setText("")
        editTextStock.setText("")
        editTextPrecio.setText("")
    } else {
        Toast.makeText(applicationContext, R.string.no_articulo_cod_msg, Toast.LENGTH_SHORT).show()
    }
}
```

3.- SQLite

- Para programar el botón de CONSULTA POR CÓDIGO también debemos crear un objeto “admin” de la clase AdministracionSQLiteOpenHelper.
- También necesitaremos el objeto “bd” de la clase **SQLiteDatabase** llamando al método getWritableDatabase (writableDatabase) del objeto admin.
- Luego creamos un objeto de la clase **Cursor** llamado “cursorConsulta” y lo inicializamos con el valor devuelto por el método **bd.rawQuery**, al que le pasamos la consulta como parámetro.
- El objeto de la clase Cursor almacena el registro (o registros) que devuelve la consulta. Llamamos al método **moveToFirst()** de la clase Cursor que retorna true en caso de existir un artículo que cumpla con la consulta.

3.- SQLite

- Para ver los datos devueltos por la consulta, llamamos al método **getString** del objeto “cursorConsulta” pasándole la posición del campo a recuperar de la SELECT (comienza a numerarse por el cero).

```
private fun consultaPorCodigo() {  
    val codigo: Int? = editTextCodigo.text.toString().toIntOrNull()  
    if(codigo == null) {  
        Toast.makeText(applicationContext, R.string.no_codigo_articulo_msq, Toast.LENGTH_SHORT).show()  
    } else {  
        val admin: AdministracionSQLiteOpenHelper =  
            AdministracionSQLiteOpenHelper(applicationContext, name: "tienda", version: 1)  
        val bd: SQLiteDatabase = admin.writableDatabase  
        val consulta: String = "SELECT descripcion, stock, precio FROM articulos WHERE codigo=$codigo"  
        val cursorConsulta: Cursor = bd.rawQuery(consulta, selectionArgs: null)  
  
        if(cursorConsulta.moveToFirst()) {  
            editTextDescripcion.setText(cursorConsulta.getString( columnIndex: 0))  
            editTextStock.setText(cursorConsulta.getString( columnIndex: 1))  
            editTextPrecio.setText(cursorConsulta.getString( columnIndex: 2))  
        } else {  
            Toast.makeText(applicationContext, R.string.no_articulo_cod_msq, Toast.LENGTH_SHORT).show()  
        }  
        bd.close()  
    }  
}
```

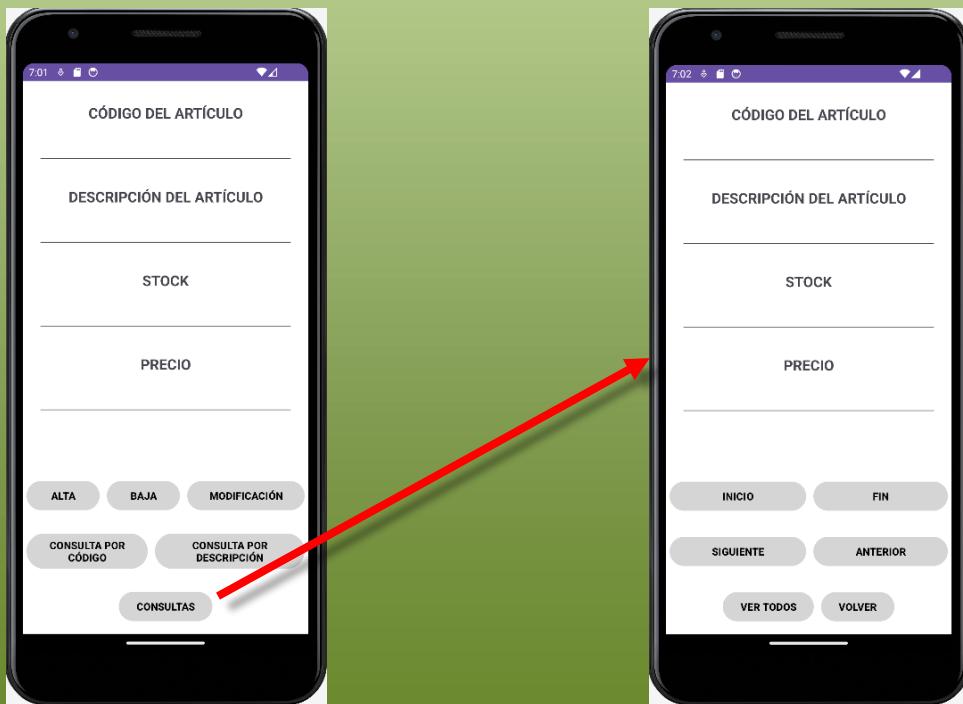
3.- SQLite

- La consulta por el campo descripción sería similar a la anterior, pero al ser un campo de tipo TEXT la cadena a buscar deberá ir entre comillas simples.
- *Fíjate en la cláusula WHERE de la SELECT, en la que se establece el campo descripción.*

```
private fun consultaPorDescripcion() {  
    val descripcion: String = editTextDescripcion.text.toString()  
    if(descripcion == "") {  
        Toast.makeText(applicationContext, R.string.no_desc_articulo_msj, Toast.LENGTH_SHORT).show()  
    } else {  
        val admin: AdministracionSQLiteOpenHelper =  
            AdministracionSQLiteOpenHelper(applicationContext, name: "tienda", version: 1)  
        val bd: SQLiteDatabase = admin.writableDatabase  
        val consulta: String = "SELECT codigo, stock, precio FROM articulos WHERE descripcion='$descripcion'"  
        val cursorConsulta: Cursor = bd.rawQuery(consulta, selectionArgs: null)  
  
        if(cursorConsulta.moveToFirst()) {  
            editTextCodigo.setText(cursorConsulta.getString(columnIndex: 0))  
            editTextStock.setText(cursorConsulta.getString(columnIndex: 1))  
            editTextPrecio.setText(cursorConsulta.getString(columnIndex: 2))  
        } else {  
            Toast.makeText(applicationContext, R.string.no_articulo_desc_msj, Toast.LENGTH_SHORT).show()  
        }  
        bd.close()  
    }  
}
```

3.- SQLite

- En caso de que la consulta nos devuelva varios registros lo ideal sería poder desplazarnos por estos. Para ello vamos a modificar nuestra aplicación y le vamos a añadir una nueva actividad:



3.- SQLite

- La consulta “**Ver todos**” se haría de la misma manera que la consulta “por código” que acabamos de hacer, necesitando los 3 objetos *admin*, *db* y *cursorConsulta*.
- En este caso, el objeto *cursorConsulta* de la clase *Cursor* lo declaramos como atributo de la clase para así poder acceder a él desde los botones de *Inicio*, *Fin*, *Siguiente* y *Anterior*, cuya funcionalidad programaremos a continuación.

3.- SQLite

```
private fun verTodos() {
    val admin: AdministracionSQLiteOpenHelper =
        AdministracionSQLiteOpenHelper(applicationContext, name: "tienda", version: 1)
    val bd: SQLiteDatabase = admin.writableDatabase
    val consulta: String = "SELECT codigo, descripcion, stock, precio FROM articulos"
    cursorConsulta = bd.rawQuery(consulta, selectionArgs: null)

    if(cursorConsulta.moveToFirst()) {
        editTextCodigo.setText(cursorConsulta.getString(columnIndex: 0))
        editTextDescripcion.setText(cursorConsulta.getString(columnIndex: 1))
        editTextStock.setText(cursorConsulta.getString(columnIndex: 2))
        editTextPrecio.setText(cursorConsulta.getString(columnIndex: 3))
    } else {
        Toast.makeText(applicationContext, R.string.no_registros_msg, Toast.LENGTH_SHORT).show()
        editTextCodigo.setText("")
        editTextDescripcion.setText("")
        editTextStock.setText("")
        editTextPrecio.setText("")
    }
    bd.close()
}
```

3.- SQLite

- Implementar los botones de *Inicio*, *Fin*, *Anterior* y *Siguiente* es muy sencillo, bastará con utilizar los métodos del objeto cursorConsulta: *moveToFirst()*, *moveToLast()*, *moveToPrevious()* y *moveToNext()*:

```
private fun inicio() {
    if(cursorConsulta.moveToFirst()) {
        editTextCodigo.setText(cursorConsulta.getString( columnIndex: 0))
        editTextDescripcion.setText(cursorConsulta.getString( columnIndex: 1))
        editTextStock.setText(cursorConsulta.getString( columnIndex: 2))
        editTextPrecio.setText(cursorConsulta.getString( columnIndex: 3))
    } else {
        Toast.makeText(applicationContext, R.string.no_registros_msa, Toast.LENGTH_SHORT).show()
        editTextCodigo.setText("")
        editTextDescripcion.setText("")
        editTextStock.setText("")
        editTextPrecio.setText("")
    }
}

private fun fin() {
    if(cursorConsulta.moveToLast()) {
        editTextCodigo.setText(cursorConsulta.getString( columnIndex: 0))
        editTextDescripcion.setText(cursorConsulta.getString( columnIndex: 1))
        editTextStock.setText(cursorConsulta.getString( columnIndex: 2))
        editTextPrecio.setText(cursorConsulta.getString( columnIndex: 3))
    } else {
        Toast.makeText(applicationContext, R.string.no_registros_msa, Toast.LENGTH_SHORT).show()
        editTextCodigo.setText("")
        editTextDescripcion.setText("")
        editTextStock.setText("")
        editTextPrecio.setText("")
    }
}
```

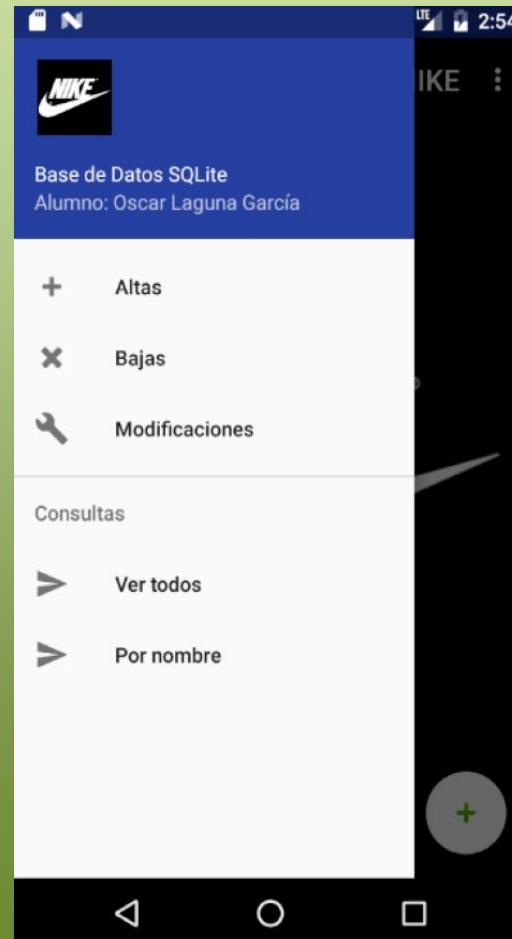
```
private fun siguiente() {
    if(cursorConsulta.moveToNext()) {
        editTextCodigo.setText(cursorConsulta.getString( columnIndex: 0))
        editTextDescripcion.setText(cursorConsulta.getString( columnIndex: 1))
        editTextStock.setText(cursorConsulta.getString( columnIndex: 2))
        editTextPrecio.setText(cursorConsulta.getString( columnIndex: 3))
    } else {
        Toast.makeText(applicationContext, R.string.ya_ultimo_registro_msa, Toast.LENGTH_SHORT).show()
    }
}

private fun anterior() {
    if(cursorConsulta.moveToPrevious()) {
        editTextCodigo.setText(cursorConsulta.getString( columnIndex: 0))
        editTextDescripcion.setText(cursorConsulta.getString( columnIndex: 1))
        editTextStock.setText(cursorConsulta.getString( columnIndex: 2))
        editTextPrecio.setText(cursorConsulta.getString( columnIndex: 3))
    } else {
        Toast.makeText(applicationContext, R.string.ya_primer_registro_msa, Toast.LENGTH_SHORT).show()
    }
}
```

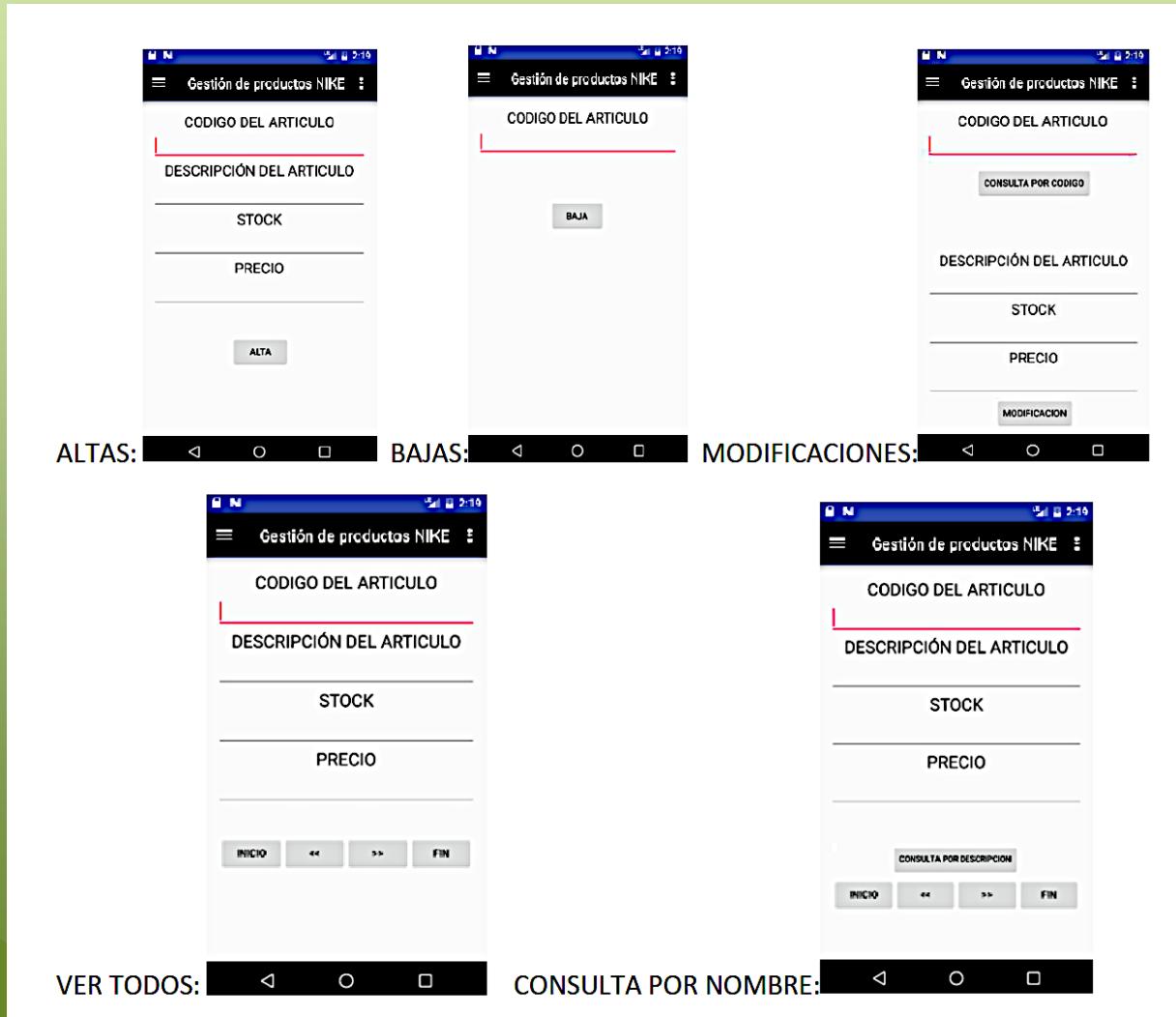
EJERCICIOS

- **Ejercicio 03 (RECOMENDADO):** Crea una pequeña aplicación Android que contenga una Base de Datos SQLite para la gestión de productos de alguna marca de ropa (Adidas, Puma, Mango, Joma, Zara, Levis, Reebook, Kelme, Lacoste...) en la que guardes su stock de artículos en una tabla que contendrá los campos: código, descripción, stock y precio.
- El programa debe permitir:
 - 1. Añadir artículos.
 - 2. Borrado de un articulo ingresando su código.
 - 3. Modificación de la descripción, stock y el precio, a partir de la búsqueda del código.
 - 4. Consulta de todos los artículos.
 - 5. Consulta de artículos por su descripción.
- Para acceder a cada una de estas opciones dispondremos de un Menú Lateral (Navigation Drawer Activity) que irá cargando cada uno de los Fragments correspondientes:

EJERCICIOS



EJERCICIOS



- Nota: Incluye en la aplicación todos los Toast que consideres necesarios.

PMDM

ALMACENAMIENTO PERMANENTE DE
DATOS



4. Conexión Firebase

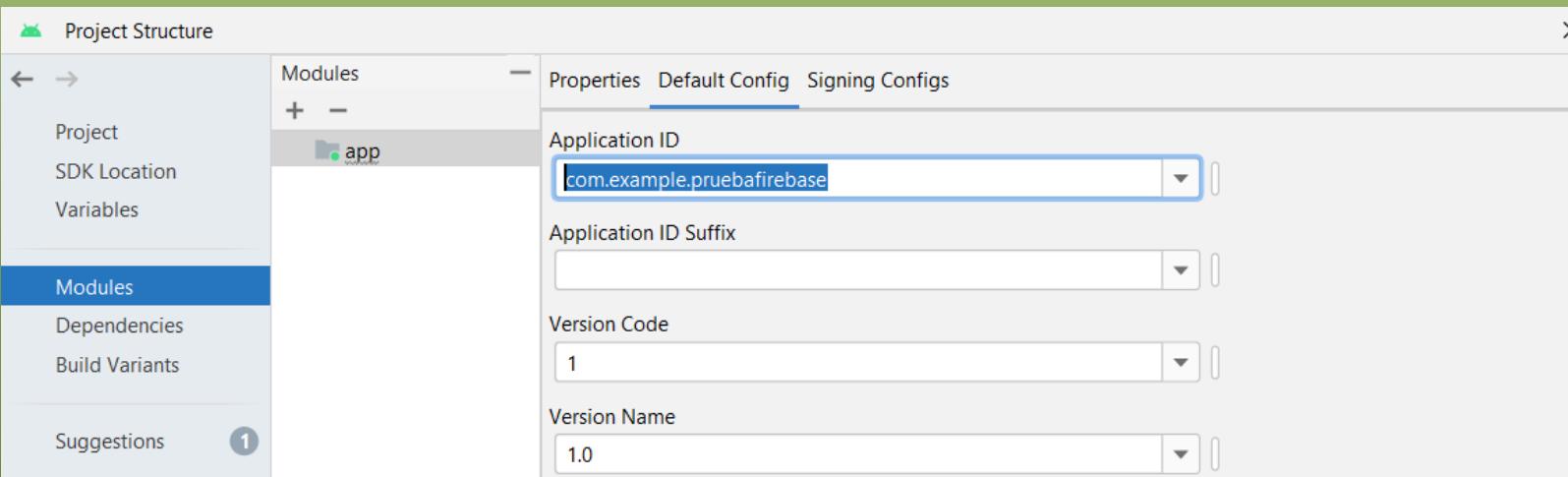
4.- Conexión Firebase

- Firebase es una plataforma para el desarrollo de aplicaciones móviles que facilita la conexión de nuestras aplicaciones con una base de datos en tiempo real, almacenando nuestros datos y permitiendo su sincronización. Sirve para aplicaciones Android, IOS y Web.
- En mayo del 2016 Google la presentó ofreciendo multitud de funcionalidades: autenticación, almacenamiento, alojamiento, notificaciones WebApps y mensajería en la nube.



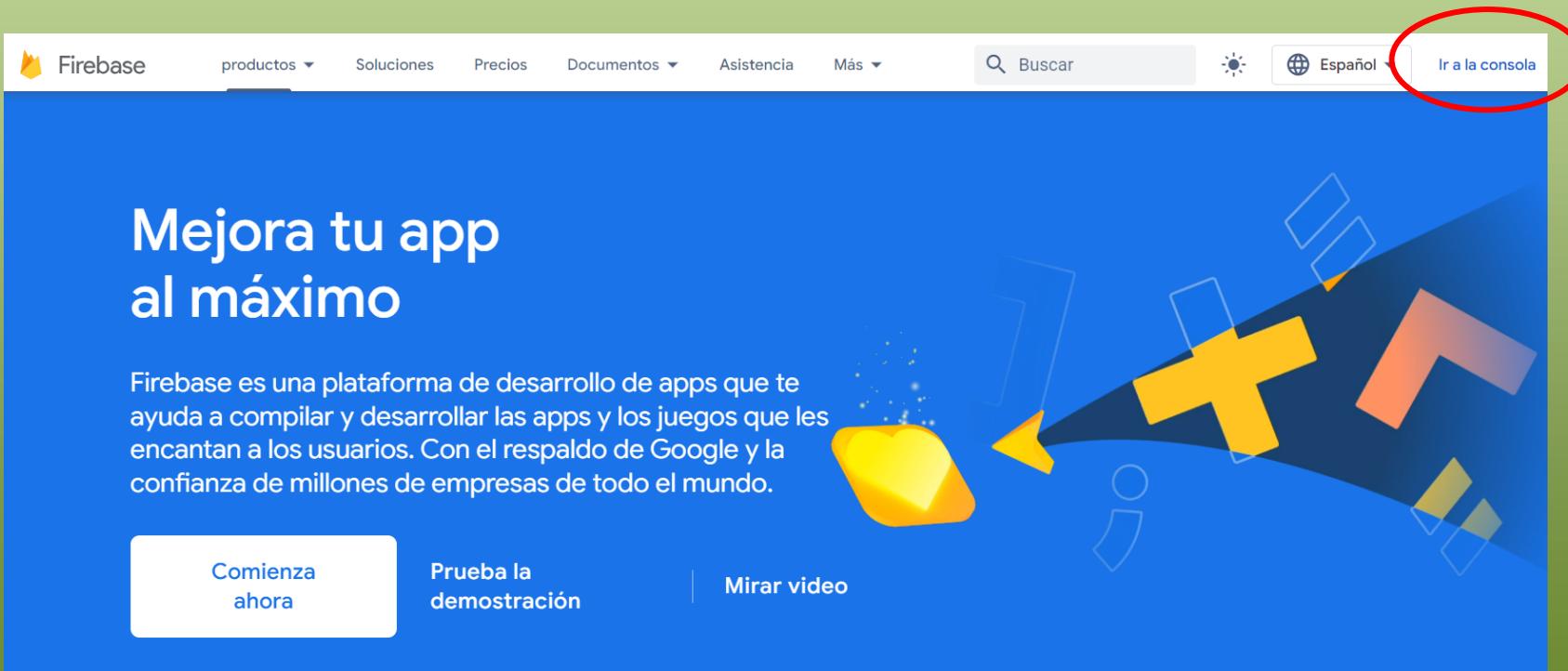
4.- Conexión Firebase

- Vamos a comenzar a utilizarla creando un proyecto en Android Studio llamado, por ejemplo, *PruebaFirebase*.
- Una vez creado el proyecto, vamos a File -> Project Structure -> Modules -> app -> Default Config y copiamos el **Application ID** de nuestra app.



4.- Conexión Firebase

- A continuación visitamos la Web de Firebase y, desde la página de bienvenida, accedemos a la consola:



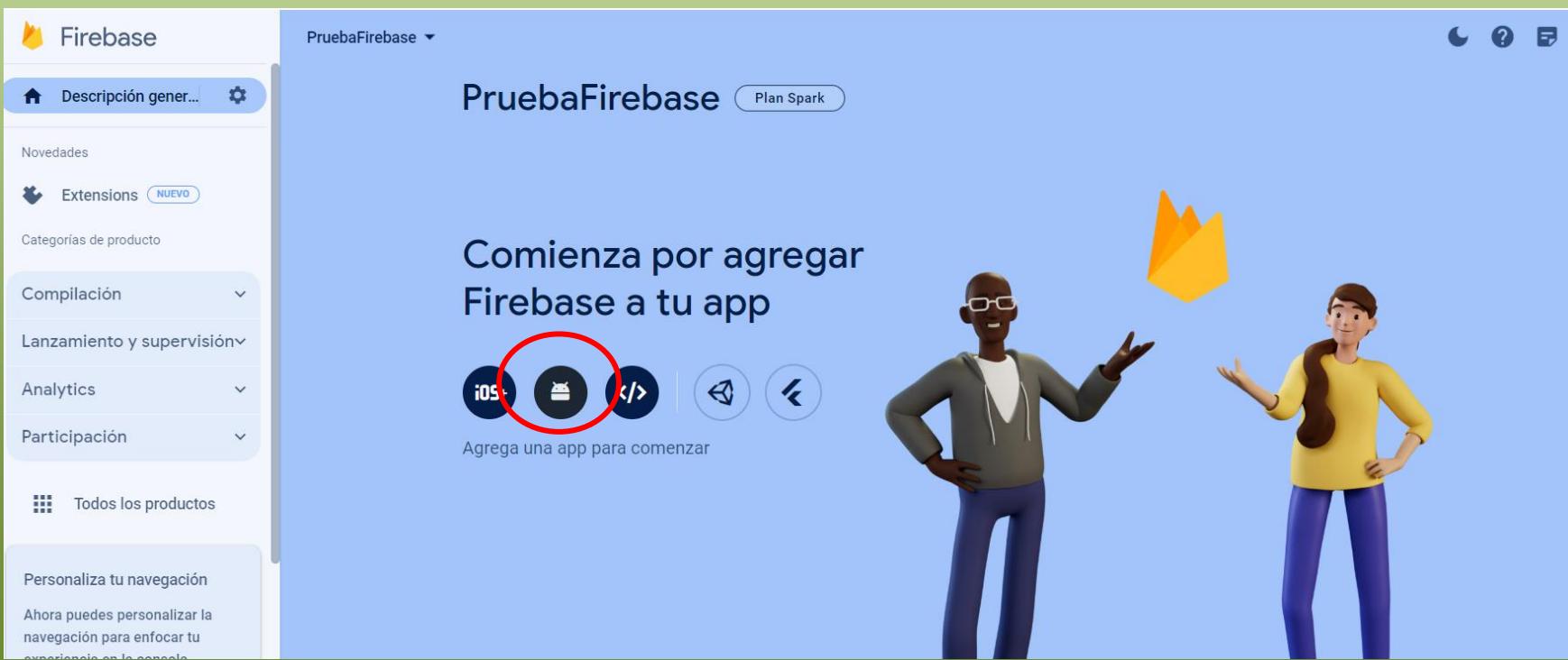
4.- Conexión Firebase

- Una vez en la consola, pulsamos en **Crear un proyecto**. Escribimos el nombre de nuestro proyecto, aceptamos los términos y condiciones, elegimos nuestro país y creamos nuestro proyecto.

The screenshot shows the Firebase console interface. On the left, there's a sidebar with a welcome message, 'Crear un proyecto' button, and 'Ver los documentos' link. The main area has a title 'Crear un proyecto(paso 1 de 3)' and a sub-instruction 'Comencemos con el nombre de tu proyecto'. A text input field contains 'PruebaFirebase'. Below the input are two buttons: 'Continuar' (blue) and 'Seleccionar recurso superior' (grey). To the right, there's a preview section showing a project named 'PruebaFirebase' with a green checkmark indicating it's ready. A large blue 'Continuar' button is at the bottom right of this section. The background features a cartoon illustration of two people working on laptops.

4.- Conexión Firebase

- Para terminar, seleccionamos **Agrega una app para comenzar** pulsando en el icono de Android:



4.- Conexión Firebase

- Pegamos nuestro *Application ID* en *Nombre del paquete Android* y el *Certificado de firma SHA-1 de depuración* para nuestra app. Posteriormente pulsamos en **Registrar app**.
- Para obtener el SHA-1 necesitamos ejecutar el comando **gradlew.bat signingReport** de Gradle, desde la ubicación de nuestro proyecto:

```
D:\workspace_android\PruebaFirebase>gradlew.bat signingReport
Welcome to Gradle 8.0!
For more details see https://docs.gradle.org/8.0/release-notes.html
Starting a Gradle Daemon, 1 incompatible Daemon could not be reused, use --status for details
> Task :app:signingReport
Variant: debug
Config: debug
Store: C:\Users\Usuario\.android\debug.keystore
Alias: AndroidDebugKey
SHA1: 59:27:59:00:38:E3:97:A5:B5:A3:1D:47:AC:D6:4F:42:3A:BB:33
SHA-256: 46:7E:64:55:ED:6F:D2:D6:89:A8:DF:A9:35:B4:51:DA:B9:D5:D5:DF:A4:57:67:5E:E9:EF:1E:D8:26:E6:0E:59
Valid until: jueves, 28 de agosto de 2053
```

4.- Conexión Firebase

x Agrega Firebase a tu app para Android

1 Registrar app

Nombre del paquete de Android ?
`com.example.pruebafirebase`

Sobrenombre de la app (opcional) ?
`Mi app para Android`

Certificado de firma SHA-1 de depuración (opcional) ?
`59:27:59:00:38:E3:97:A5:3F:B5:A3:1D:47:AC:D6:4F:42`

● Obligatoria para Dynamic Links y el Acceso con Google o la asistencia con un número de teléfono en Auth. Puedes editar las claves SHA-1 en Configuración.

Registrar app

- El siguiente paso es descargar el archivo **google-services.json** y colocarlo en nuestro proyecto como se indica en la web de Firebase. Pulsamos en **Siguiente**.

4.- Conexión Firebase

Agreea Firebase a tu app para Android

✓ Registrar app
Nombre del paquete de Android: com.example.pruebafirebase

2 Descargar y, luego, agregar el archivo de configuración a continuación | [Unity](#) [C++](#)

[Descargar google-services.json](#)

Cambia a la vista Proyecto en Android Studio para ver el directorio raíz de tu proyecto.

Mueve el archivo `google-services.json` descargado al directorio raíz de tu módulo (nivel de app).

The screenshot shows the "Agrega Firebase a tu app para Android" (Add Firebase to your Android app) wizard. Step 2 is active, instructing the user to download the "google-services.json" file and move it to the app module's root directory in Android Studio. A blue arrow points from the "google-services.json" download button to the "app" module in the Android Studio Project Explorer. Another blue arrow points from the "google-services.json" icon to the file listed in the project tree.

[Siguiente](#)

4.- Conexión Firebase

- El último paso consiste en añadir un par de líneas en el **gradle del proyecto** y en el **gradle de la aplicación**, y después sincronizar.

1. Para que los SDK de Firebase puedan acceder a los valores de configuración de `google-services.json`, necesitas el complemento Gradle de los servicios de Google.

DSL de Kotlin (`build.gradle.kts`) Groovy (`build.gradle`)

Agrega el complemento como una dependencia a tu archivo `build.gradle.kts` de nivel de proyecto:

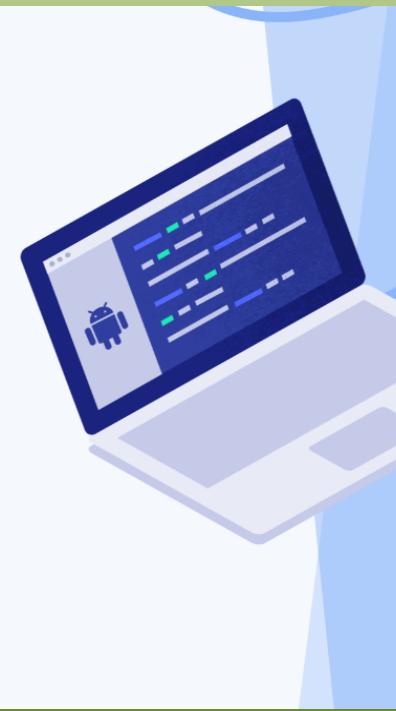
Archivo de Gradle de nivel de raíz (nivel de proyecto) (<project>/`build.gradle.kts`):

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

2. Luego, en el archivo `build.gradle.kts` del **módulo (nivel de app)**, agrega los complementos `google-services` y cualquier SDK de Firebase que quieras usar en tu app:

Archivo de Gradle del módulo (nivel de app) (<project>/<app-module>/`build.gradle.kts`):

```
plugins {
    id("com.android.application")
    // Add the Google services Gradle plugin
    id("com.google.gms.google-services")
}
```



4.- Conexión Firebase

- A continuación vamos a añadir las siguientes bibliotecas de la SDK de Firebase al build.gradle.kts del módulo app de nuestro proyecto, para empezar a crear nuestra base de datos:
 - **BoM:** Se encarga de descargar las últimas versiones compatibles existentes en la SDK de Firebase para las aplicaciones Android.
 - **Realtime Database:** Biblioteca para usar la base de datos de Firebase.
- Sincronizamos el gradle para descargar las dependencias.

The screenshot shows the Firebase console interface. On the left, there's a sidebar with sections like 'productos', 'Soluciones', 'Precios', 'Docs', 'Comunidad', 'Ayuda', and a search bar. The main area has a heading 'Aspectos básicos' and a 'Comience con Firebase' section with 'Agregar Firebase a una aplicación' and 'Plataformas Apple (iOS+)'. Below that is a 'Android' section with 'Realtime Database' highlighted by a red box. To the right, there's a table listing various Firebase products and their dependencies. The 'Realtime Database' row is also highlighted with a red box. The table columns include the product name, dependency name, and version.

Visualización de In-App Messaging	com.google.firebaseio:firebase-inappmessaging-display-ktx	20.3.5
Instalaciones de Firebase	com.google.firebaseio:firebase-installations-ktx	17.1.4
API de Firebase ML Model Downloader	com.google.firebaseio:firebase-ml-modelldownloader-ktx	24.1.3
Performance Monitoring	com.google.firebaseio:firebase-perf-ktx	20.4.1
Complemento de Performance Monitoring	com.google.firebaseio:perf-plugin	1.4.2
Realtime Database	com.google.firebaseio:firebase-database-ktx	20.2.2
Remote Config	com.google.firebaseio:firebase-config-ktx	21.4.1

The screenshot shows an Android Studio project structure with a 'build.gradle.kts (app)' file open. The code includes the 'dependencies' block where the 'Realtime Database' dependency is added. A red box highlights this addition. The code also includes comments about the BoM and the Realtime Database dependency.

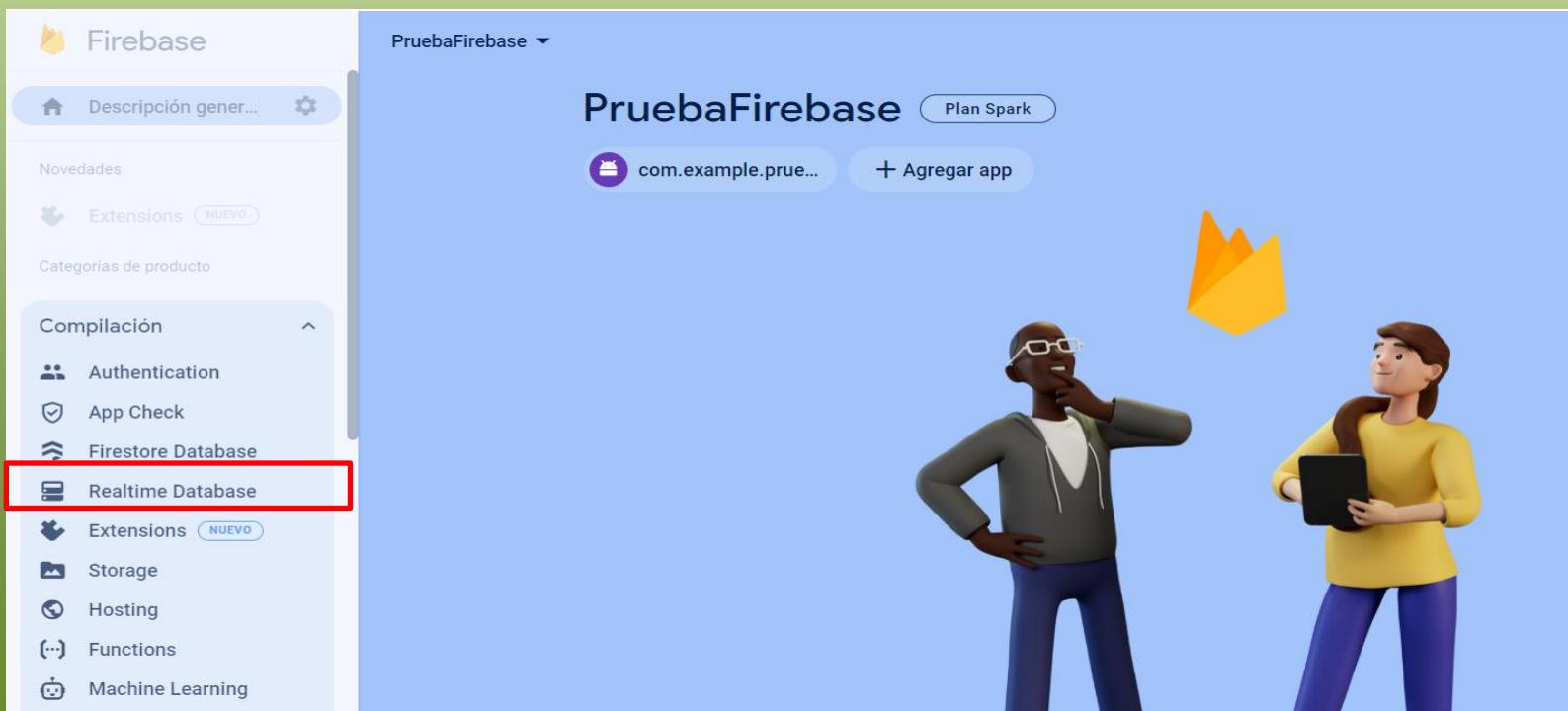
```
dependencies { this.DependencyHandlerScope
    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("com.google.android.material:material:1.11.0")
    implementation("androidx.constraintlayout:constraintlayout:2.1.4")
    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
}

//Firebase
//BoM: Se encarga de descargar las últimas versiones compatibles existentes en la SDK de Firebase para las aplicaciones Android
implementation(platform("com.google.firebaseio:firebase-bom:32.7.1"))

//Realtime Database
implementation("com.google.firebaseio:firebase-database-ktx")
```

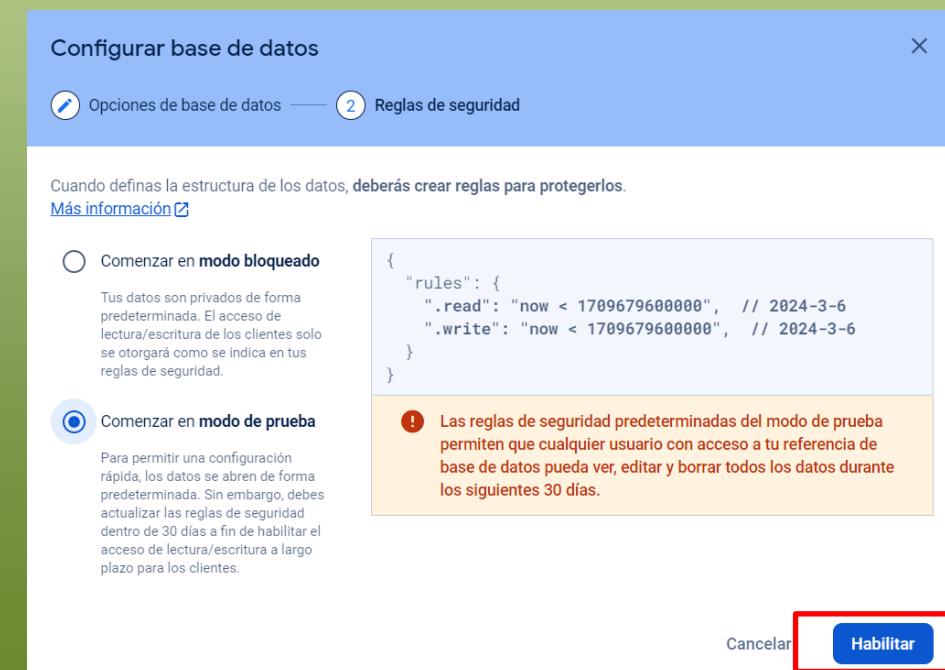
4.- Conexión Firebase

- Vamos a crear una Realtime Database en FireBase. Para ello, seleccionamos nuestro proyecto “PruebaFirebase” y elegimos la opción **Realtime Database** en el menú **Compilación** de la parte izquierda de la ventana:



4.- Conexión Firebase

- Pulsamos en **Crear una base de datos** y seleccionamos **Comenzar en modo prueba**, para configurar reglas de escritura y lectura que no necesiten autenticación (ya que aún no la hemos implementado):



4.- Conexión Firebase

- Accedemos a la pestaña **Reglas** de Realtime Database y establecemos las reglas de lectura y escritura a “true” (acceso público), para poder acceder sin autenticación desde nuestra app Android:

The screenshot shows the Firebase Realtime Database rules editor. The left sidebar lists projects: 'Descripción general...', 'Realtime Database' (selected), 'Extensions (NUEVO)', 'Categorías de producto', 'Compilación', 'Lanzamiento y supervisión', 'Analytics', and 'Participación'. The main area is titled 'Realtime Database' and has tabs for 'Datos', 'Reglas' (selected), 'Copias de seguridad', 'Uso', and 'Extensões'. A warning message in a pink box says: '⚠ Tus reglas de seguridad están definidas como públicas, por lo que cualquiera puede robar, modificar o borrar información de tu base de datos'. Below is the JSON security rules code:

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

A red box highlights the entire 'rules' block in the JSON code.

4.- Conexión Firebase

- La bases de datos en Firebase son NO RELACIONALES, Y NO PERMITEN CONSULTAS SQL, ya que almacenan la información en un sencillo fichero **JSON**. Además, Firebase en su versión gratuita, solo permiten 100 conexiones simultáneas.
- Sabiendo esto, vamos a ver cómo escribiríamos en la base de datos Firebase desde nuestra aplicación Android:

4.- Conexión Firebase

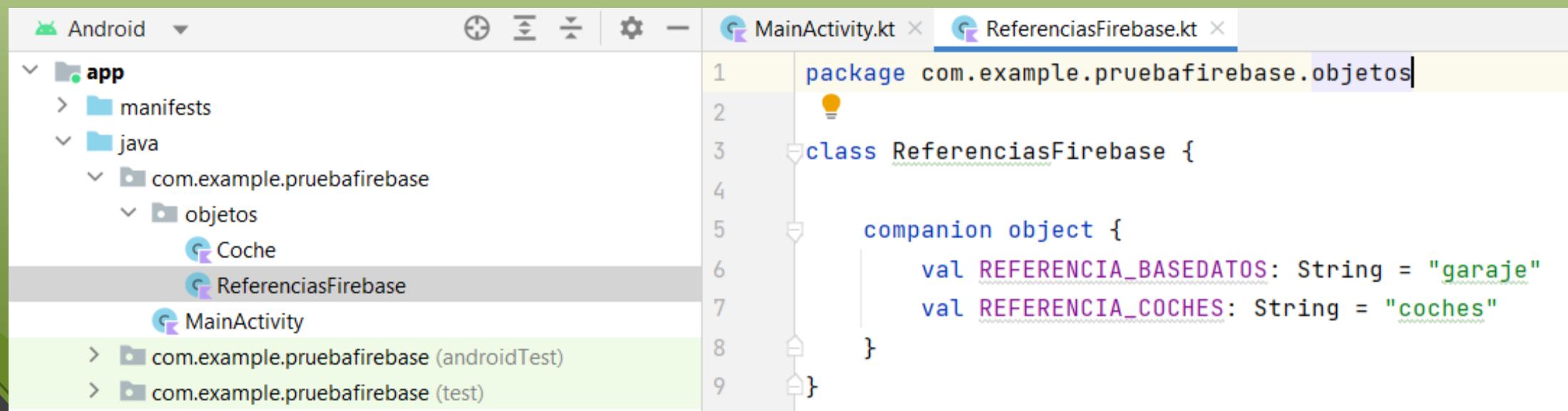
- La captura muestra el JSON resultado que queremos obtener en Firebase, una vez escritos los datos desde nuestra app.
- Por buscar un símil a las bases de datos SQL, “garaje” sería la base de datos, “coches” sería la tabla, el identificador de objeto (key) sería la clave, y los atributos del objeto (marca, modelo, numeroPuertas y velocidadMaxima) serían los campos de la tabla.

The screenshot shows the Firebase Realtime Database interface. At the top, there's a navigation bar with tabs: Datos (selected), Reglas, Copias de seguridad, Uso, and Extensiones. Below the navigation bar, the title "PruebaFirebase" is followed by "Realtime Database". The main area displays a hierarchical JSON structure under the path "garaje/coches". The first node has keys "marca", "modelo", "numeroPuertas", and "velocidadMaxima" with values "Renault", "Megane", 5, and 180 respectively. The second node has the same structure with values "Ford", "Focus", 5, and 200. On the left side, there's a URL link: "https://pruebafirebase-f2b32-default.firebaseio.com".

```
graph TD; Root["https://pruebafirebase-f2b32-default.firebaseio.com"] --> garaje[garaje]; garaje --> coches[coches]; coches --> Renault["-NpvR5CBe8guEv0SAyfS | marca: Renault, modelo: Megane, numeroPuertas: 5, velocidadMaxima: 180"]; coches --> Ford["-NpvR5CPpr6DRAyunQEM | marca: Ford, modelo: Focus, numeroPuertas: 5, velocidadMaxima: 200"];
```

4.- Conexión Firebase

- Comenzaremos creando una clase donde guardaremos las referencias a la base de datos Firebase. Esto es, el nombre de la base de datos y el nombre de las tablas:

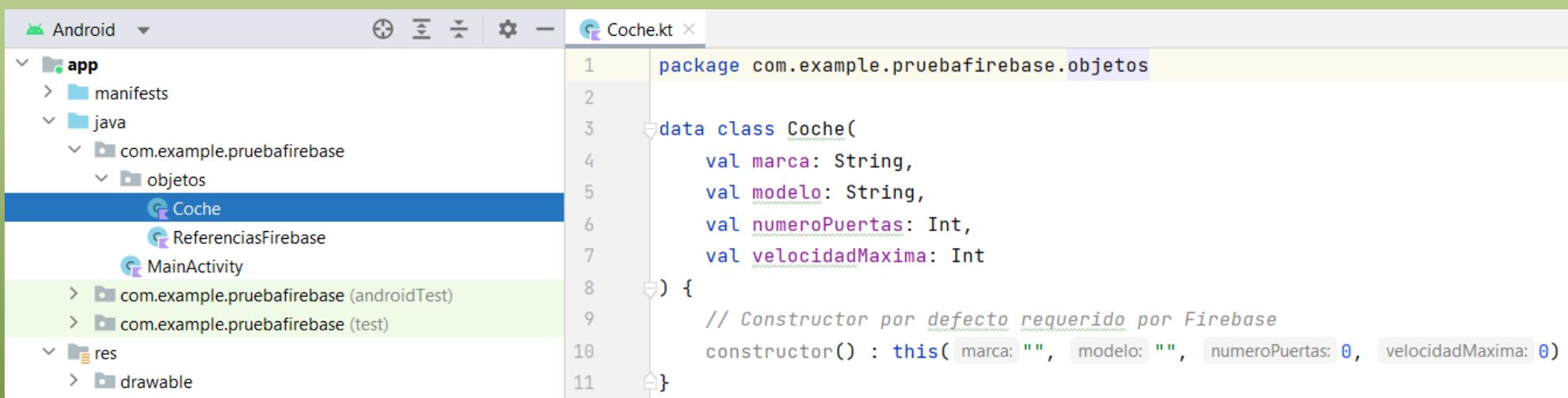


The screenshot shows the Android Studio interface. On the left, the project structure is displayed under the 'app' module. It includes 'manifests', 'java' (with a package named 'com.example.pruebafirebase' containing 'objetos', 'ReferenciasFirebase', and 'MainActivity'), and test packages. The right side shows the code editor for 'ReferenciasFirebase.kt'. The code defines a class 'ReferenciasFirebase' with a companion object containing two string constants: 'REFERENCIA_BASEDATOS' set to 'garaje' and 'REFERENCIA_COCHES' set to 'coches'.

```
package com.example.pruebafirebase.objetos
class ReferenciasFirebase {
    companion object {
        val REFERENCIA_BASEDATOS: String = "garaje"
        val REFERENCIA_COCHES: String = "coches"
    }
}
```

4.- Conexión Firebase

- También necesitaremos una clase para instanciar los objetos que luego volcaremos a la base de datos. Para ello, implementamos una **data class** con el nombre **Coche**:



The screenshot shows the Android Studio interface. On the left, the project structure is displayed under the 'app' module. It includes 'manifests', 'java' (with packages 'com.example.pruebafirebase' and 'objetos'), 'res', and 'drawable'. Inside 'objetos', there are three files: 'Coche' (selected), 'ReferenciasFirebase', and 'MainActivity'. The 'Coche' file is open in the main editor. The code is as follows:

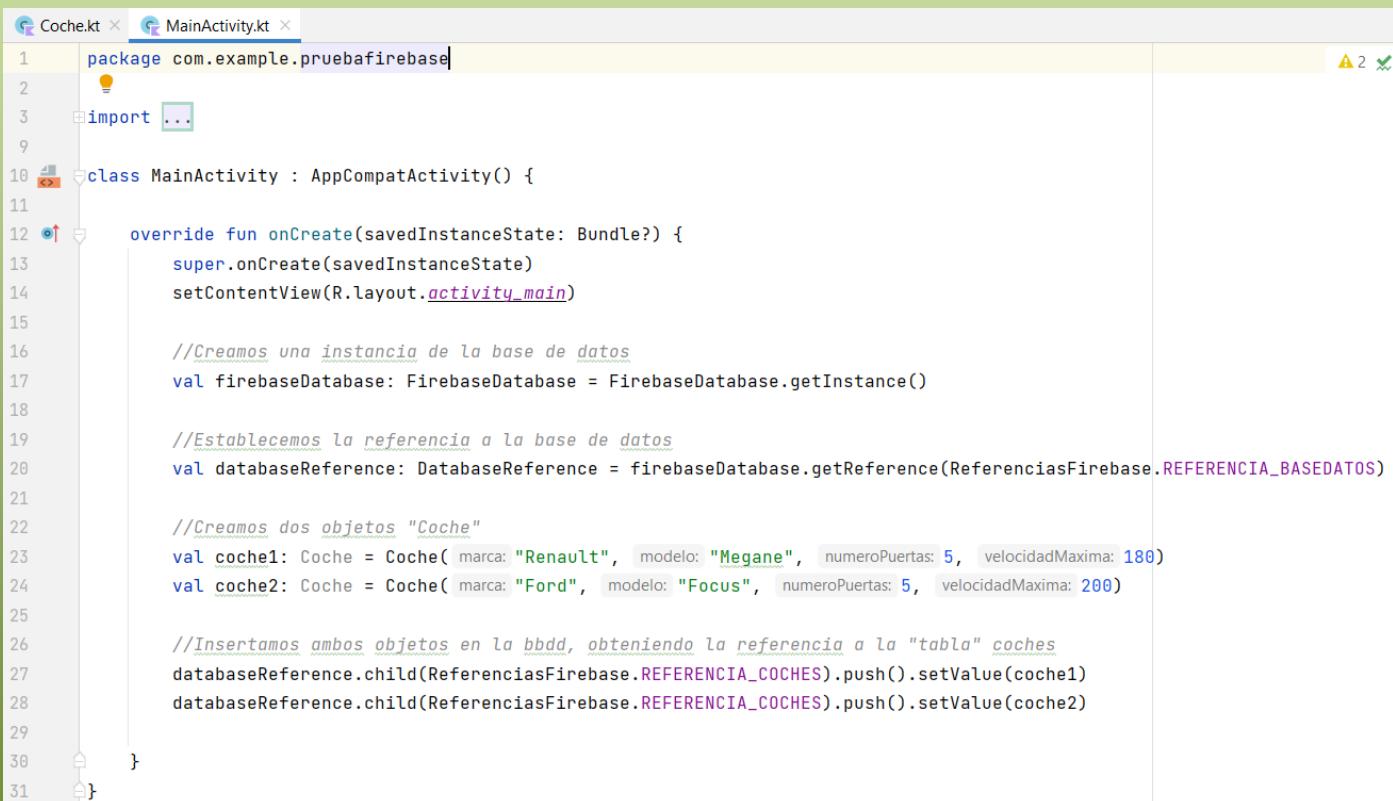
```
package com.example.pruebafirebase.objetos

data class Coche(
    val marca: String,
    val modelo: String,
    val numeroPuertas: Int,
    val velocidadMaxima: Int
) {
    // Constructor por defecto requerido por Firebase
    constructor() : this("", "", 0, 0)
}
```

- NOTA:** *Para que las librerías de Firebase puedan mapear los datos que leen de la base de datos en nuestra data class, se debe implementar el constructor por defecto sin argumentos de la clase.*

4.- Conexión Firebase

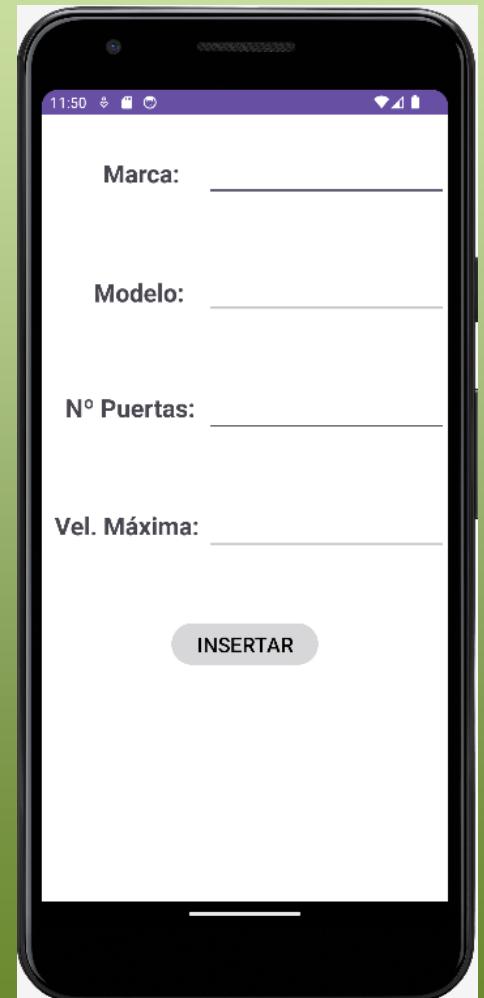
- Implementamos el código en el método `onCreate()` de la `MainActivity.kt` y, al ejecutar nuestra aplicación, veremos como en la base de datos de Firebase se crean los objetos indicados en nuestra app:



```
Coche.kt x MainActivity.kt x
1 package com.example.pruebafirebase
2
3 import ...
4
5 class MainActivity : AppCompatActivity() {
6
7     override fun onCreate(savedInstanceState: Bundle?) {
8         super.onCreate(savedInstanceState)
9         setContentView(R.layout.activity_main)
10
11         //Creamos una instancia de la base de datos
12         val firebaseDatabase: FirebaseDatabase = FirebaseDatabase.getInstance()
13
14         //Establecemos la referencia a la base de datos
15         val databaseReference: DatabaseReference = firebaseDatabase.getReference(ReferenciasFirebase.REFERENCIA_BASEDATOS)
16
17         //Creamos dos objetos "Coche"
18         val coche1: Coche = Coche(marca: "Renault", modelo: "Megane", numeroPuertas: 5, velocidadMaxima: 180)
19         val coche2: Coche = Coche(marca: "Ford", modelo: "Focus", numeroPuertas: 5, velocidadMaxima: 200)
20
21         //Insertamos ambos objetos en la bbdd, obteniendo la referencia a la "tabla" coches
22         databaseReference.child(ReferenciasFirebase.REFERENCIA_COCHES).push().setValue(coche1)
23         databaseReference.child(ReferenciasFirebase.REFERENCIA_COCHES).push().setValue(coche2)
24
25     }
26
27 }
```

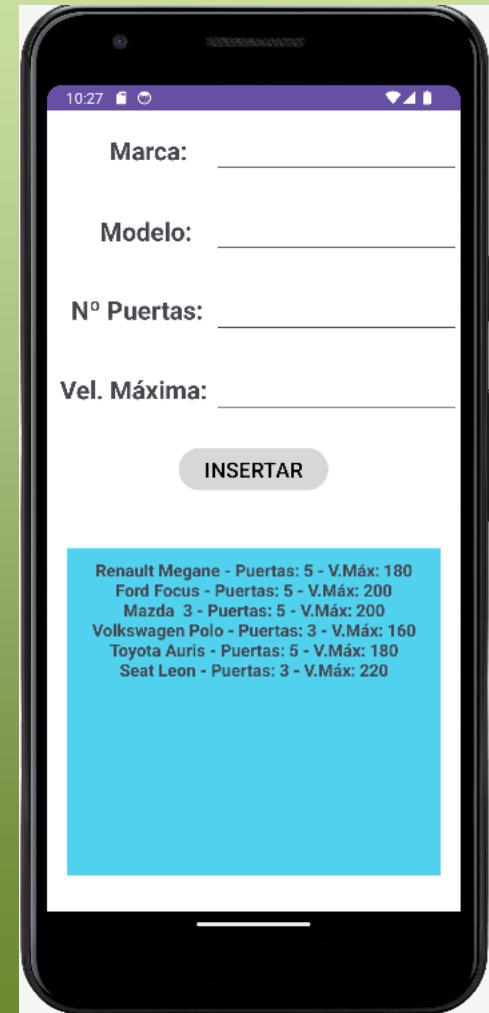
4.- Conexión Firebase

- A continuación vamos a mejorar nuestra aplicación, de tal forma que el usuario introduzca los datos en los editText y que, al pulsar en el botón “INSERTAR”, almacene los nuevos datos en Firebase, limpie los editText y muestre un Toast indicando que la operación de inserción se ha realizado correctamente.



4.- Conexión Firebase

- Para terminar, vamos a añadir un `textView` que se rellene con los datos de la tabla “coches” de Firebase.
- Además, implementaremos un `addValueEventListener` que esté siempre escuchando. De esta manera, cada vez que haya un cambio en la base de datos de Firebase, se nos actualizará el `textView` aplicando los nuevos cambios.



4.- Conexión Firebase

- Creamos una variable en la clase **MainActivity** que implemente el **addValueEventListener**.
- En el método **onDataChange** de este evento, realizamos un **foreach** que recorra todos los coches de la base de datos (se guardan en el objeto **dataSnapshot**) y mostramos los datos en el **textView**.

```
//Listener que escucha los cambios que se producen en la bbdd
private val valueEventListener = object : ValueEventListener {

    override fun onDataChange(dataSnapshot: DataSnapshot) {
        //Variable donde almacenamos el texto con los datos de todos los coches
        var datosCoches: String = ""
        //Se recorre el listado de coches obteniendo todos sus datos
        for (cocheSnapshot in dataSnapshot.children) {
            val cocheAux: Coche? = cocheSnapshot.getValue(Coche::class.java)
            if(cocheAux != null) {
                datosCoches = datosCoches + cocheAux.marca + " " + cocheAux.modelo +
                    " - Puertas: " + cocheAux.numeroPuertas + " - V.Máx: " + cocheAux.velocidadMaxima + "\n"
            }
        }
        //Se actualiza el textView con los datos de todos los coches
        textViewMostrar.setText(datosCoches)
    }

    override fun onCancelled(databaseError: DatabaseError) {
        //Error de acceso a la base de datos
        Toast.makeText(applicationContext, "Error de acceso", Toast.LENGTH_SHORT).show()
    }
}
```

4.- Conexión Firebase

- En el método onCreate() de la MainActivity.kt, utilizamos la referencia a la base de datos para establecer el.addValueEventListener a la tabla “coches”:

```
//Creamos una instancia de la base de datos
firebaseDatabase = FirebaseDatabase.getInstance()
//Establecemos la referencia a la base de datos
databaseReference = firebaseDatabase.getReference(ReferenciasFirebase.REFERENCIA_BASEDATOS)

//Asignamos el valueEventListener a la "tabla" coches de la base de datos
databaseReference.child(ReferenciasFirebase.REFERENCIA_COCHES).addValueEventListener(valueEventListener)
```

4.- Conexión Firebase

- Si quisieramos presentar los datos de una manera más elegante, lo más adecuado sería mostrarlos en un ListView o en un RecycledView.

EJERCICIOS

- **Ejercicio 04 (RECOMENDADO):** Crea una pequeña aplicación Android que se conecte a una Base de Datos Firebase para la gestión de tu colección de videojuegos en la que guardes su título, plataforma, empresa desarrolladora y el año.
- El programa debe permitir lo mismo que hemos visto en el ejemplo de clase.
 - 1. Añadir videojuegos a Firebase.
 - 2. Consulta de todos los videojuegos (utiliza un ListView o un Recycled View).
- Para acceder a cada una de estas opciones dispondremos de un **Menú Lateral (Navigation Drawer)** que irá cargando cada uno de los Fragments correspondientes.