

CURSO DE TÉCNICO EN DESARROLLO NATIVO SOBRE PLATAFORMAS ANDROID

EOI – Escuela de Organización Industrial

TEMA 5. I/O, DB e interoperabilidad

CONTENIDOS

1. Shared Preferences
2. Ficheros
3. Operaciones con ficheros
4. Bases de datos con Room
5. Comunicación cliente-servidor con Firebase
6. Notificaciones Push en Firebase

SHARED PREFERENCES

- **SharedPreferences** es un objeto Java que apunta a un fichero
- El fichero contiene parejas **clave-valor**, con valores de **tipo primitivo o String**
- A través de **SharedPreferences** podemos leer y escribir en dicho fichero
- Se utiliza para almacenar pequeñas colecciones de parejas clave-valor interesantes para nuestra aplicación

doc: <https://developer.android.com/training/data-storage/shared-preferences#java>

SHARED PREFERENCES

- Ejemplos de parejas **clave-valor**:
 - “username” : “manelizzard”
 - “toolbar_color” : “red”
 - “background_color” : “black”
 - “sound_enabled” : true
 - “vibration_enabled” : true
 - “push_notifications_enabled” : false

SHARED PREFERENCES

- Se pueden obtener tres tipos diferentes de SharedPreferences:
 - Varios ficheros con identificador único
 - Fichero genérico y exclusivo para una Activity concreta
 - Fichero genérico accesible desde cualquier punto de nuestra aplicación

doc: <https://developer.android.com/training/data-storage/shared-preferences#java>

SHARED PREFERENCES

- Se pueden obtener **varios ficheros de *preferences***, identificados por nombre

```
SharedPreferences sharedPref = getSharedPreferences(  
    getString(R.string.basic_preference_file),  
    Context.MODE_PRIVATE);
```

```
<!-- Preference files in strings.xml -->  
<string name="basic_preference_file">  
com.android.teaching.miprimeraapp.BASIC_PREFERENCE_FILE  
</string>
```

SHARED PREFERENCES

- *Context.MODE_PRIVATE* - El fichero creado **es solo accesible desde nuestra aplicación**
- Los identificadores de los distintos ficheros de SharedPreferences deben ser únicos e identificables de tu aplicación. Para ello, añadimos el nombre del paquete.

SHARED PREFERENCES

- Se pueden obtener un fichero *genérico* para nuestra **Activity**
- No requiere nombre y pertenece **exclusivamente** a la Activity donde se invoca

```
SharedPreferences sharedPref =  
    getPreferences(Context.MODE_PRIVATE);
```


SHARED PREFERENCES

- Si se quiere un fichero genérico accesible desde cualquier punto de la aplicación, se utiliza **getDefaultSharedPreferences** del objeto **PreferenceManager**

```
SharedPreferences sharedPref =  
PreferenceManager.getDefaultSharedPreferences(this);
```

SHARED PREFERENCES

- Para escribir en un objeto **SharedPreferences**:
 1. Obtener una instancia de **Editor**
 2. Escribir la pareja clave-valor deseada con **put...**
 3. Invocar **editor.apply()**

```
SharedPreferences sharedPref =  
    getPreferences(Context.MODE_PRIVATE);  
SharedPreferences.Editor editor = sharedPref.edit();  
editor.putString("username", usernameValue);  
editor.apply();
```

SHARED PREFERENCES

- Para leer un valor de **SharedPreferences**:
 1. Utilizar los métodos *get...* del objeto `SharedPreferences`
- Al igual que los extras en **Intent**, al obtener un valor de **SharedPreferences** podemos indicar un valor por defecto.

```
SharedPreferences sharedPref =  
    getPreferences(Context.MODE_PRIVATE);  
String value = sharedPref  
    .getString("username", "no-username");
```

SHARED PREFERENCES

- Para borrar un valor de **SharedPreferences**:
 1. Obtener el objeto **Editor**
 2. Invocar **remove** con el nombre del valor que queremos borrar
 3. Invocar **editor.apply()**

```
SharedPreferences sharedPref =  
    getPreferences(Context.MODE_PRIVATE);  
SharedPreferences.Editor editor = sharedPref.edit();  
editor.remove("username");  
editor.apply();
```

PRÁCTICA

1. En **LoginActivity**, cuando el usuario pulsa el botón “**Login**”, guardar su nombre de usuario en **SharedPreferences**
 - Podéis utilizar cualquiera de los tres tipos de **SharedPreferences**
2. Cada vez que se vuelva a mostrar la pantalla **LoginActivity**, recuperar el nombre de usuario de **SharedPreferences** y escribirlo automáticamente en el campo *username*

PRÁCTICA

1. En **ProfileActivity**, guardar todos los valores de los campos en **SharedPreferences** cuando la Activity pasa por su método **onStop()**
2. Cada vez que se vuelva a mostrar la pantalla **ProfileActivity**, recuperar los valores de los campos y escribirlos automáticamente
 - Hacerlo en el método **onStart()**

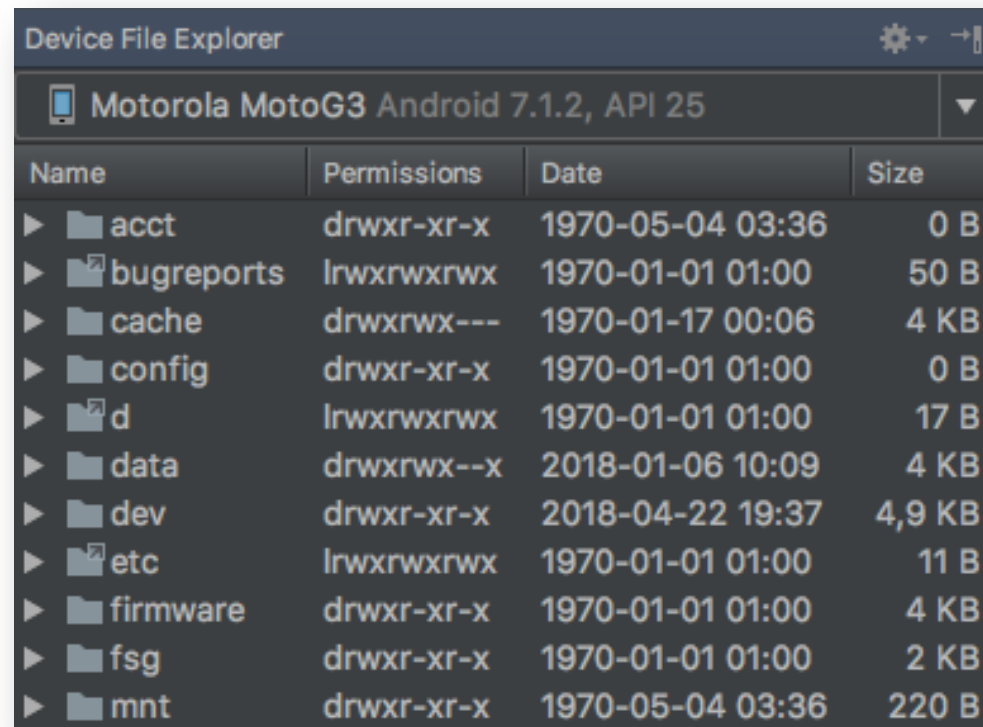
FICHEROS

- Android almacena ficheros de igual manera que cualquier otra plataforma con sistema de disco
- **File** representa un objeto Java apuntando a un fichero en el disco
- Los ficheros sirven para almacenar grandes cantidades de datos de principio a fin, como imágenes o contenido descargado de Internet

doc: <https://developer.android.com/training/data-storage/files>

FICHEROS

- Android Studio permite explorar el sistema de ficheros de nuestro dispositivo en
 - **View > Tool Windows > Device File Explorer**



Name	Permissions	Date	Size
▶ folder acct	drwxr-xr-x	1970-05-04 03:36	0 B
▶ folder bugreports	lrwxrwxrwx	1970-01-01 01:00	50 B
▶ folder cache	drwxrwx---	1970-01-17 00:06	4 KB
▶ folder config	drwxr-xr-x	1970-01-01 01:00	0 B
▶ folder d	lrwxrwxrwx	1970-01-01 01:00	17 B
▶ folder data	drwxrwx--x	2018-01-06 10:09	4 KB
▶ folder dev	drwxr-xr-x	2018-04-22 19:37	4,9 KB
▶ folder etc	lrwxrwxrwx	1970-01-01 01:00	11 B
▶ folder firmware	drwxr-xr-x	1970-01-01 01:00	4 KB
▶ folder fsg	drwxr-xr-x	1970-01-01 01:00	2 KB
▶ folder mnt	drwxr-xr-x	1970-05-04 03:36	220 B

FICHEROS

- **¡WARNING!** En función del fabricante del dispositivo, la ruta de ficheros reservada para una aplicación puede variar. Por ello, **nunca utilizar rutas absolutas**
 - ~~Guardar en... “sdcard/test/ruta_publica”~~
- **Utilizar siempre los métodos facilitados por Android para obtener las rutas de ficheros**
 - *getFilesDir(), getCacheDir(), getExternalStoragePublicDirectory(), etc...*

FICHEROS

- Áreas de almacenamiento
 - **Internal:** sirve para almacenar ficheros que únicamente son accesibles desde nuestra aplicación.
 - **External:** sirve para almacenar ficheros que son accesible desde cualquier otra aplicación, o ficheros que queremos compartir con otras aplicaciones o vía USB.

FICHEROS

Internal storage:

- It's always available.
- Files saved here are accessible by only your app.
- When the user uninstalls your app, the system removes all your app's files from internal storage.

Internal storage is best when you want to be sure that neither the user nor other apps can access your files.

External storage:

- It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
- It's world-readable, so files saved here may be read outside of your control.
- When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from `getExternalFilesDir()`.

External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

FICHEROS

- **Almacenamiento interno.** Existen dos directorios para el almacenamiento interno de nuestra aplicación:
 - **Internal Directory:** directorio básico donde almacenar nuestros ficheros.
 - **Cache Directory:** directorio de *cache* para guardar ficheros temporales. Android puede borrar estos ficheros en función de los recursos del dispositivo.

FICHEROS

- **Almacenamiento interno.** Para obtener un objeto tipo **File** con los directorios:
 - **Internal Directory** -> *getFilesDir()*

```
File directorioInterno = getFilesDir();
```

- **Cache Directory** -> *getCacheDir()*

```
File directorioCache = getCacheDir();
```

FICHEROS

- Algunas acciones disponibles en **File API**:
 - **File API** es la propia de Java
 - *exists()* -> determina si el fichero existe o no
 - *getName()* -> obtiene el nombre del fichero
 - *getAbsolutePath()* -> obtiene la ruta absoluta del fichero
 - *mkdir()* -> crea el directorio representado en **File**
 - *makedirs()* -> igual que *mkdir()*, pero creando todos los directorios intermedios (si fuese necesario)
 - . . .

doc: <https://developer.android.com/reference/java/io/File>

PRÁCTICA

1. En nuestra Activity principal, en el método **onCreate**, imprimir la ruta absoluta del almacenamiento interno de nuestra aplicación.
2. Hacer lo mismo para la ruta de nuestro almacenamiento interno de cache.

```
File directorioInterno = ...;  
File directorioCache = ...;  
Log.d("ListActivity", "Interno: " + directorioInterno...);  
Log.d("ListActivity", "Cache: " + directorioCache...);
```

FICHEROS

- **Almacenamiento interno.** Abrir un fichero.
 - **Internal Directory** -> *getFilesDir()*

```
File myFile = new File(getFilesDir(), "nombre_fichero.txt");
```

- **Cache Directory** -> *getCacheDir()*

```
File myFile = new File(getCacheDir(), "nombre_fichero.txt");
```


FICHEROS

- **Almacenamiento externo**
 - **Puede existir o no** en el dispositivo del usuario. Por ejemplo, el usuario puede extraer su tarjeta SD del dispositivo.
 - Hasta la API 18 (Android 4.3), **se requería un permiso especial en AndroidManifest**. Si nuestra aplicación solo soporta de Android 4.4 en adelante, no se requieren permisos para leer/escribir en el almacenamiento externo

FICHEROS

- **Permisos de lectura/escritura** (Solo para API menor de 19)

```
<manifest ... >
    <uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```

FICHEROS

- Siempre debemos comprobar si el almacenamiento externo existe y está disponible para escribir

```
// Checks if external storage is available for read and write
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```

FICHEROS

- Si no podemos escribir, comprobar al menos que existe y se puede leer

```
// Checks if external storage is available to at least read
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state))
    {
        return true;
    }
    return false;
}
```

PRÁCTICA

1. Crear en **ProfileActivity** los dos métodos de comprobación de estado del almacenamiento interno:
 - Comprobar si existe el almacenamiento y se puede utilizar
 - Comprobar si se puede escribir en él o, por el contrario, es de solo lectura
2. Imprimir el resultado en Logcat con **Log.d()**

FICHEROS

- **Almacenamiento externo.** Existen dos tipos de directorios en el almacenamiento externo:
 - **Público:** guarda un fichero público y accesible por otras aplicaciones en el dispositivo.
 - **Privado:** el fichero sigue siendo público y accesible, pero no tiene sentido fuera de nuestra aplicación. Al desinstalar la aplicación, este directorio se borrará por completo.

FICHEROS

- **Almacenamiento externo.** Para guardar un fichero en el almacenamiento **público** externo se utiliza el método:
 - **getExternalStoragePublicDirectory(type)**
 - El parámetro **type** indica el tipo de fichero que queremos almacenar y así organizarlos de forma lógica:
 - Environment.DIRECTORY_MUSIC
 - Environment.DIRECTORY_PICTURES
 - ...

FICHEROS

- Ejemplo

```
File file = Environment  
    .getExternalStoragePublicDirectory(  
        Environment.DIRECTORY_PICTURES);
```


FICHEROS

- **Almacenamiento externo.** Para guardar un fichero en el almacenamiento **privado** externo se utiliza el método:
 - **getExternalFilesDir(type)**
 - El parámetro **type** indica el mismo tipo que en los ficheros públicos, pero si, en este caso, indicamos **null**, el método devolverá la raíz del directorio

PRÁCTICA

1. En **ProfileActivity**, en el método **onCreate**, imprimir la ruta absoluta del almacenamiento externo de nuestra aplicación, siempre y cuando exista y sea accesible.

```
File directorioExterno = ...;  
Log.d("ListActivity", "Externo: " + directorioExterno...);
```

FICHEROS

- **Almacenamiento externo.** Abrir un fichero.
 - **External Directory** -> *getExternalFilesDir(type)*

```
File myFile = new File(getExternalFilesDir(null),  
                        "nombre_fichero.txt");
```

- **Cache Directory** -> *getExternalCacheDir()*

```
File myFile = new File(getExternalCacheDir(),  
                        "nombre_fichero.txt");
```

OPERACIONES CON FICHEROS

- Borrar fichero:
 - delete() -> retorna un *boolean* para informar si el fichero se ha borrado correctamente

```
File myFile = new File(getExternalFilesDir(null), "test.txt");  
myFile.delete();
```

OPERACIONES CON FICHEROS

- Escribir un fichero:
 - Se utiliza el objeto Java **FileOutputStream**

```
String filename = "myfile";
String fileContents = "Hello world!";
FileOutputStream outputStream;

try {
    outputStream = openFileOutput(filename,
                                   Context.MODE_PRIVATE);
    outputStream.write(fileContents.getBytes());
    outputStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

OPERACIONES CON FICHEROS

- Leer un fichero (línea a línea)

```
File file = new File(getFilesDir(),"fichero.txt");
StringBuilder text = new StringBuilder();

try {
    BufferedReader br = new BufferedReader(new
                                           FileReader(file));

    String line;

    while ((line = br.readLine()) != null) {
        text.append(line);
    }
    br.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

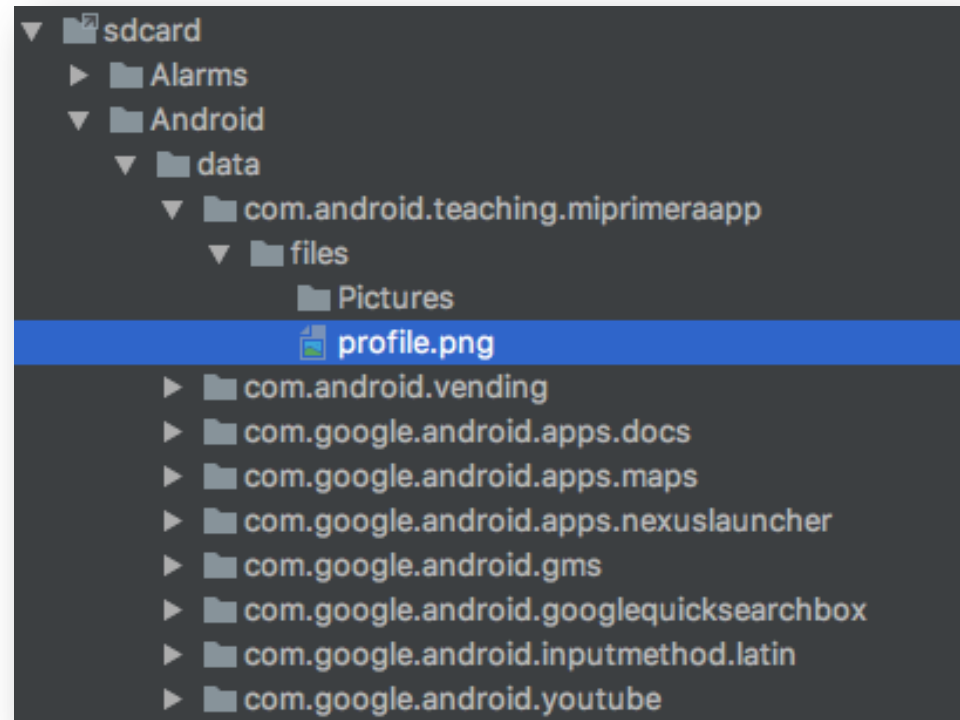
OPERACIONES CON FICHEROS

- Obtener espacio disponible
 - **getFreeSpace()** -> obtiene el espacio disponible en un directorio
 - **getTotalSpace()** -> obtiene el espacio total de almacenamiento

```
File externalDirectory = getExternalFilesDir(null);  
externalDirectory.getFreeSpace();  
externalDirectory.getTotalSpace();
```

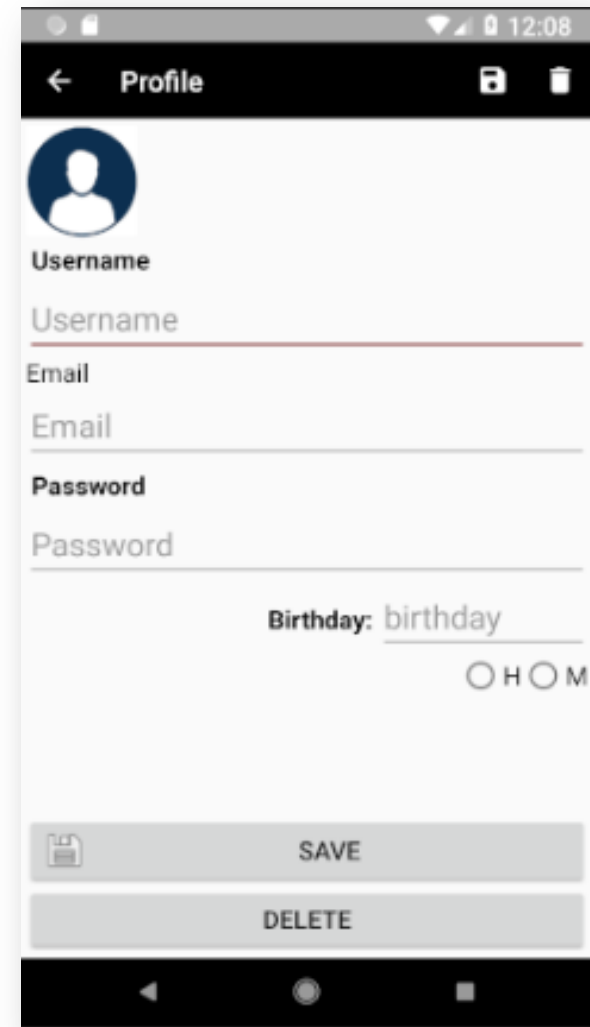
PRÁCTICA

1. Añadir una imagen de perfil en el directorio externo de vuestra aplicación:
**sdcard/Android/
data/
com.android.teaching.miprimeraapp/files**



PRÁCTICA

1. Crear un **ImageView** en **ProfileActivity** y mostrar la imagen almacenada en la tarjeta SD (siempre la misma)



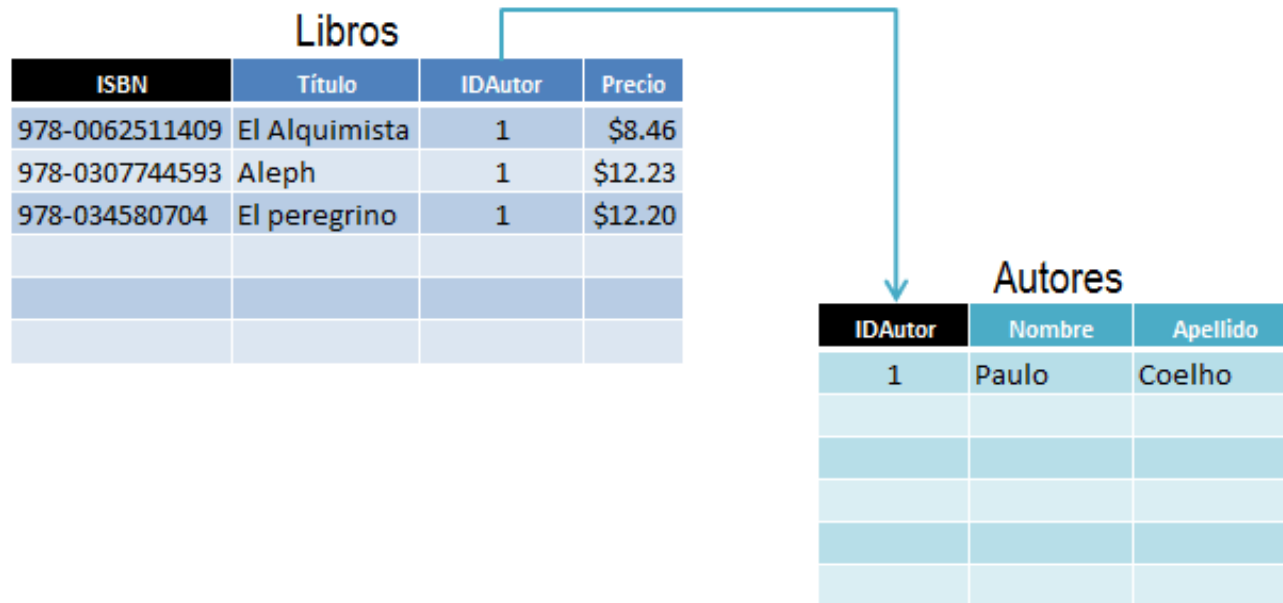
PRÁCTICA

1. Cómo añadir una imagen desde fichero a un **ImageView**:

```
File imgFile = new ...;  
if(imgFile.exists()) {  
    ImageView myImage= findViewById(R.id.profile_image_view);  
    myImage.setImageURI(Uri.fromFile(imgFile));  
}
```

BASES DE DATOS

- **¿Que es una base de datos?**
 - Colección de datos organizados en un conjunto de tablas formalmente descritas. Existen relaciones entre tablas y datos.



BASES DE DATOS

- **SQL** y **SQLite** son lenguajes estándar para acceder y manipular bases de datos.
- **SQLite** es una versión reducida de **SQL**.
- Android gestiona bases de datos mediante **SQLite**.
- En un entorno con conexión a Internet casi permanente, las bases de datos locales representan un sistema de *cache* de datos, permitiendo al usuario utilizar la aplicación navegar mientras está sin conexión

BASES DE DATOS

- **Ejemplos SQLite**

- SELECT * FROM [Customers]
- SELECT * FROM [Customers] WHERE City IS "London"
- SELECT * FROM [OrderDetails] ORDER BY Quantity
- SELECT * FROM [Orders],[Shippers] WHERE Shippers.ShipperID IS "1"
- INSERT INTO [Shippers] VALUES (4, "MRW", "1234")
- DELETE FROM [Shippers] WHERE ShipperID IS 4

- Para aprender más:

- https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all

ROOM

- **Room** es una librería que Google nos ofrece para manejar, de forma sencilla y rápida, nuestras bases de datos locales.
- Aún así, podemos utilizar la ~~vieja~~ manera de usar SQLite en Android:
<https://developer.android.com/training/data-storage/sqlite>

doc: <https://developer.android.com/training/data-storage/room/>

ROOM

- **Room** utiliza **annotations** para identificar todos los atributos y métodos que nos permitirán gestionar la base de datos.
- Estas anotaciones se escriben **sobre la clase, sobre el atributo o sobre el método** al que se aplican, y van precedidas por el carácter **@**

ROOM

- Ejemplo de anotaciones en Room:

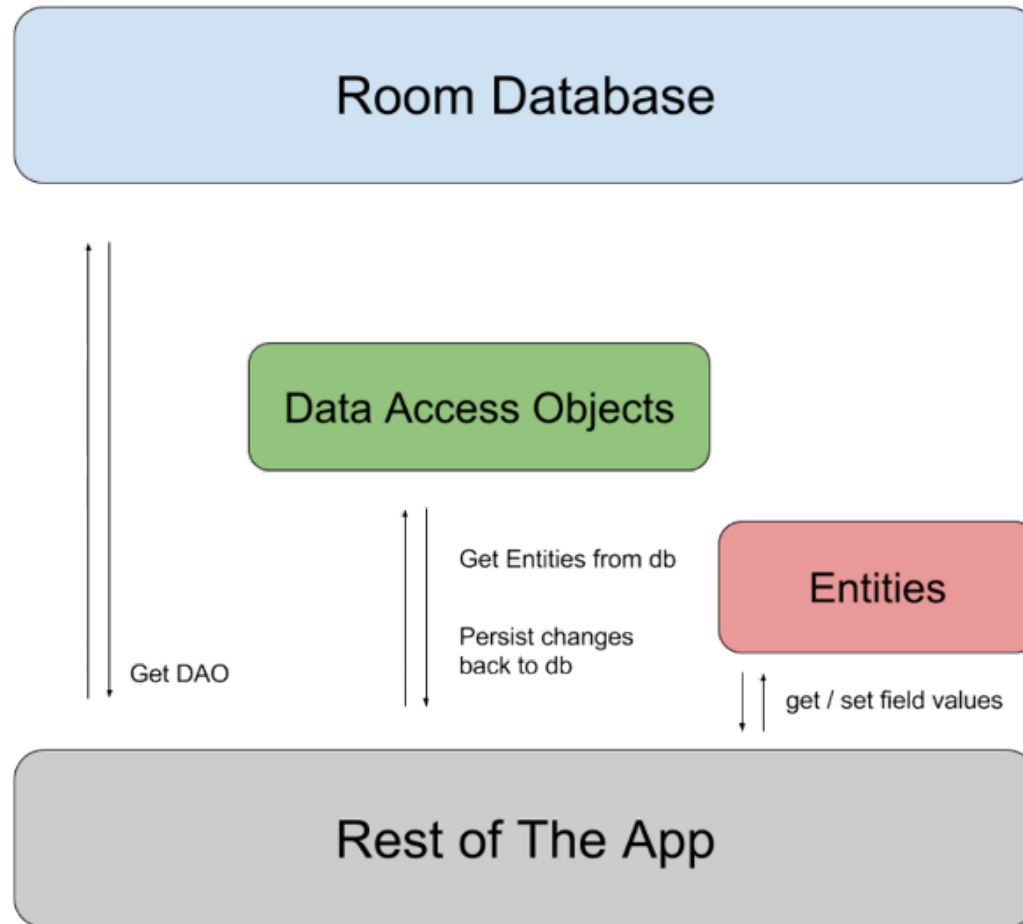
```
@Entity
public class User {
    @PrimaryKey(autoGenerate = true)
    private int uid;

    @ColumnInfo(name = "username")
    private String username;
}
```


ROOM

- **Room** contiene tres componentes principales:
 - **Database**: Contiene la base de datos en sí y es la vía principal de acceso a ella.
 - **Entity**: Representa una tabla de la base de datos.
 - **DAO**: Contiene los métodos necesarios para acceder a la base de datos.

ROOM



ROOM

- **Room Entity**

- Representa una tabla de la base de datos
- La clase se anota con **@Entity** y los atributos y métodos pueden también contener anotaciones:
 - **@PrimaryKey**. Indica la clave primaria de la tabla. Se puede indicar que sea autogenerada. Siempre debe anotarse con **@NonNull**, debido a que la clave primaria nunca puede estar vacía.
 - **@ColumnInfo**. Podemos indicar el nombre de la columna de la tabla.

ROOM

```
@Entity  
public class User {  
    @PrimaryKey(autoGenerate = true)  
    @NotNull  
    private int uid;  
  
    @ColumnInfo(name = "username")  
    private String username;  
  
    @ColumnInfo(name = "email")  
    private String email;  
  
    @ColumnInfo(name = "password")  
    private String password;  
  
    // ¡¡¡IMPORTANTE! getters y setters  
}
```

ROOM

- **Room Dao (Data Access Object)**
 - Por cada **Entity**, existe un **DAO** que nos ofrece métodos para acceder a la tabla que representa
 - Un **DAO** se define como una **interfaz Java** con anotaciones en cada uno de sus métodos, ya sea para:
 - Obtener datos de la tabla (SELECT)
 - Insertar nuevos datos (INSERT)
 - Eliminar datos (DELETE)

ROOM

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE username IS :username")
    User findByUsername(String username);

    @Insert
    void insert(User user);

    @Delete
    void delete(User user);
}
```

ROOM

- **Room Database**
 - Clase abstracta que hereda de **RoomDatabase**
 - Debe anotarse con **@Database** e indicar las *entities* y la versión de la base de datos.

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

ROOM

- **Room Database**

- Por cada **DAO**, se define un método que nos devolverá el **DAO** correspondiente, con los que podremos acceder a las tablas de la base de datos.

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```


ROOM

- Para obtener un objeto **AppDatabase** en nuestra aplicación:
 - **Room.databaseBuilder().build()**
 - *allowMainThreadQueries()* -> permite ejecutar accesos a la base de datos en el **UIThread**.
¡SOLO HASTA QUE VEAMOS RxAndroid!

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),  
    AppDatabase.class, "database-name")  
    .allowMainThreadQueries()  
    .build();
```

ROOM

- Añadiendo **Room** a nuestra aplicación:
 - Añadir las dependencias en **app/build.gradle**
 - Sincronizar el proyecto

[Sync Now](#)

```
dependencies {  
    . . .  
    def room_version = "1.1.0"  
    implementation  
        "android.arch.persistence.room:runtime:$room_version"  
    annotationProcessor  
        "android.arch.persistence.room:compiler:$room_version"  
    . . .  
}
```

PRÁCTICA

1. Crear una **Entity** llamada **User** con los siguientes atributos:

- username - String (**clave primaria no auto generada**)
- email - String
- password - String
- age - String (ej: “21”, “7/5/1989”)
- gender - String (ej: “H” o “M”)

2. Añadir los **getters** y **setters**

PRÁCTICA

1. Crear un **DAO** llamado **UserDao** que tenga los siguientes métodos:
 - **User findUserByUsername(String username);**
 - *Obtiene un usuario de la base de datos según su 'username' (SELECT)*
 - **void insert(User user);**
 - *Inserta un usuario en la base de datos*
 - **void delete(User user);**
 - *Borra un usuario de la base de datos*

PRÁCTICA

1. Crear la clase abstracta **AppDatabase** que herede de **RoomDatabase** y nos permita obtener un **UserDao** para manipular la tabla **User**.
 - La versión de la base de datos es 1.

ROOM

- Para **insertar** un objeto en la base de datos, debemos usar el **DAO** correspondiente
 - **Ejemplo:** Para insertar una nueva fila en la tabla **User**, debemos utilizar ***userDao.insert(User user);***

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),  
    AppDatabase.class, "database-name")  
    .allowMainThreadQueries()  
    .build();
```

```
User user = new User();  
.  
.  
db.userDao().insert(user);
```

ROOM

- Al insertar, debemos **garantizar que la clave primaria no se repetirá**. Si se repite, el método *insert()* lanzará una excepción de tipo **SQLiteConstraintException**
- Es conveniente capturar la excepción con **try {} catch() {}** para evitar sorpresas

ROOM

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
    AppDatabase.class, "database-name")
    .allowMainThreadQueries()
    .build();

try {
    User user = new User();
    .
    .
    .
    db.userDao().insert(user);
} catch (SQLiteConstraintException ex) {
    // Algún error ha ocurrido al insertar
}
```


PRÁCTICA

1. En **ProfileActivity**, cuando el usuario pulse el botón **“Save”**:
 - Crear un objeto de tipo **User** con todos los valores que el usuario ha escrito en los campos.
 - Guardar el objeto **User** en la base de datos recién creada a través de **UserDao**

PRÁCTICA

1. En **LoginActivity**, cuando el usuario pulse el botón “**Login**”:
 - Obtener de la base de datos el objeto **User** cuyo *username* se corresponde con el login del usuario.
 - Si el **User** no existe en la base de datos, mostrar un error tipo toast: “**Login failed**”
 - **Si el usuario existe**, comprobar que las contraseñas son iguales (la del campo de texto y la del usuario de la base de datos). Si son iguales, proceder a abrir **ProfileActivity**. Si no, mostrar un error en password.

PRÁCTICA

1. En **ProfileActivity**, cuando la Activity se inicia (onCreate):
 - Buscar si tenemos en **SharedPreferences** el nombre de usuario guardado.
 - Si lo tenemos, buscar y obtener el objeto **User** de la base de datos que corresponde con dicho nombre de usuario.
 - Si existe el usuario en la base de datos, rellenar los campos de texto con los valores del objeto **User**

ROOM

- Para **borrar** un objeto en la base de datos, debemos usar el **DAO** correspondiente
 - **Ejemplo:** Para borrar un **User**, debemos utilizar ***userDao.delete(User user);***

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),  
    AppDatabase.class, "database-name")  
    .allowMainThreadQueries()  
    .build();
```

```
User user = new User();  
.  
.  
db.userDao().delete(user);
```

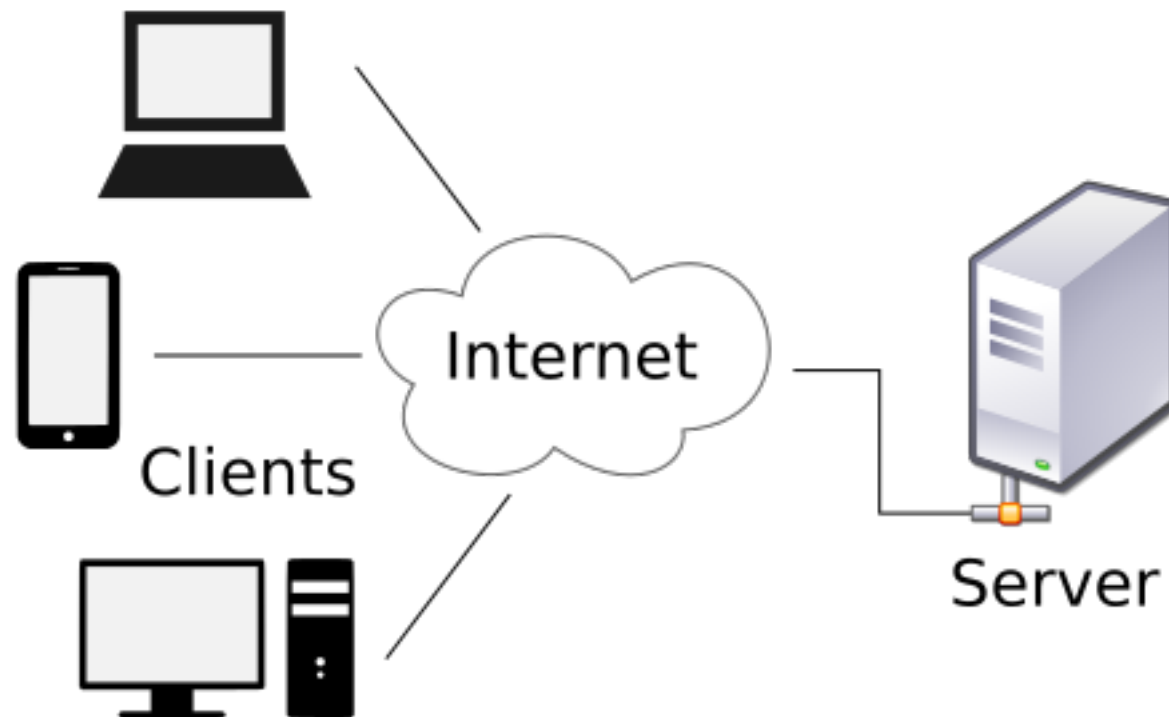
PRÁCTICA

1. En **ProfileActivity**, cuando el usuario pulse el botón “**Delete**” y después del dialogo de confirmación:
 - Borrar de la base de datos el objeto **User** que previamente se ha cargado (si existe).
 - Si no existe, no hacer nada.

ARQUITECTURA CLIENTE - SERVIDOR

- Hoy día, la gran mayoría de aplicaciones móviles se nutren de una **base de datos alojada en un servidor en la nube**
- Esta base de datos es **muy dinámica**, y los dispositivos (clientes) actualizan su contenido según los últimos datos de la base de datos

ARQUITECTURA CLIENTE - SERVIDOR

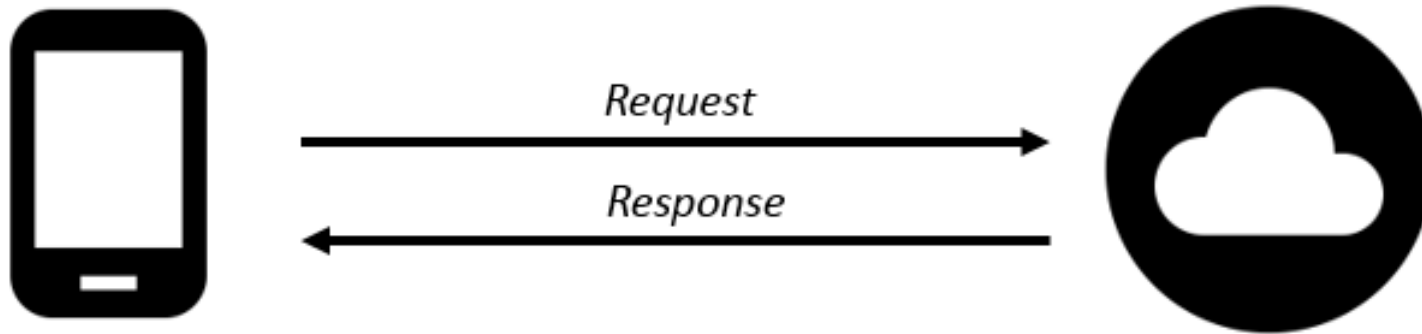


ARQUITECTURA CLIENTE - SERVIDOR

- Cuando una aplicación cliente obtiene datos de un servidor, estos son guardados **en memoria**, no en disco. Por tanto, al cerrar la aplicación, los datos no se persisten.
- Se pueden persistir los datos explícitamente con bases de datos locales, ficheros, etc...

ARQUITECTURA CLIENTE - SERVIDOR

- Flujo de comunicación



HTTP
Hyper Text Type Protocol

ARQUITECTURA CLIENTE - SERVIDOR

- Protocolo HTTP (**H**ypertext **T**ransfer **P**rotocol):
 - Protocolo de comunicación cliente-servidor
 - Se pueden realizar varias acciones:
 - GET – Sirve para pedir información al servidor
 - POST – Sirve para guardar datos en el servidor
 - DELETE – Sirve para borrar algún recurso del servidor
 - ...

doc: https://www.w3schools.com/tags/ref_httpmethods.asp

ARQUITECTURA CLIENTE - SERVIDOR

- **HTTP GET**

- Sirve para pedir datos a un servidor
- Puede contener parámetros. Estos se especifican en la misma URL como parejas **clave-valor**
- **Ejemplo:**

<http://pokeapi.co/api/v2/pokemon/mew/>

<http://pokeapi.co/api/v2/pokemon/1/>

<http://pokeapi.co/api/v2/item/poke-ball/>

ARQUITECTURA CLIENTE - SERVIDOR

- **HTTP POST**

- Sirve para pedir guardar datos en un servidor
- Los datos a guardar se especifican en el **body** de la petición
- **Ejemplo:**

```
POST /test/demo_form.php HTTP/1.1  
Host: w3schools.com  
name1=value1&name2=value2
```

ARQUITECTURA CLIENTE - SERVIDOR

- Flujo de comunicación cliente – servidor
 1. El cliente realiza una petición al servidor vía HTTP o HTTPS (GET, POST, etc...)
 2. El servidor obtiene los datos necesarios de su base de datos y devuelve un mensaje al cliente, normalmente en formato XML o JSON
 3. El cliente obtiene la respuesta, la analiza, *parsea* y la convierte en **objetos modelo** para ser utilizados por la interfaz

ARQUITECTURA CLIENTE - SERVIDOR

- Las respuestas HTTP de un servidor contienen un **código** que determina el estado de la respuesta:
 - **200 – Ok.** Significa que todo ha ido bien
 - **403 – Forbidden.** No se ha podido autenticar la petición y el servidor la rechaza.
 - **404 – Not Found.** El servidor no encuentra el recurso requerido por la petición
 - **500 – Internal Server Error.**
 - . . .

doc: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html>

ARQUITECTURA CLIENTE - SERVIDOR

- La información recibida desde servidor puede ser en formato XML, JSON, etc...
 - **La más común es JSON (JavaScript Object Notation)**
 - Es un formato ligero
 - Es sencillo y fácil de entender a simple vista
 - Es independiente de la plataforma: es puramente texto
 - Al recibir esta información, es necesario transformarla en el modelo de nuestra aplicación

doc: https://www.w3schools.com/js/js_json_intro.asp

FIREBASE

- **Firestore** es una plataforma y herramienta ofrecida por Google para desarrollar de forma rápida, sencilla y eficaz una aplicación móvil con un servidor en la nube.



Firestore

FIREBASE

- **Firestore** nos proporciona herramientas como:
 - **Realtime Database**
 - **Cloud Messaging**
 - **Crashlytics**
 - **Autenticación**
 - **Almacenamiento**
 - **Hosting**
 - **Test Lab**
 - ...

FIREBASE

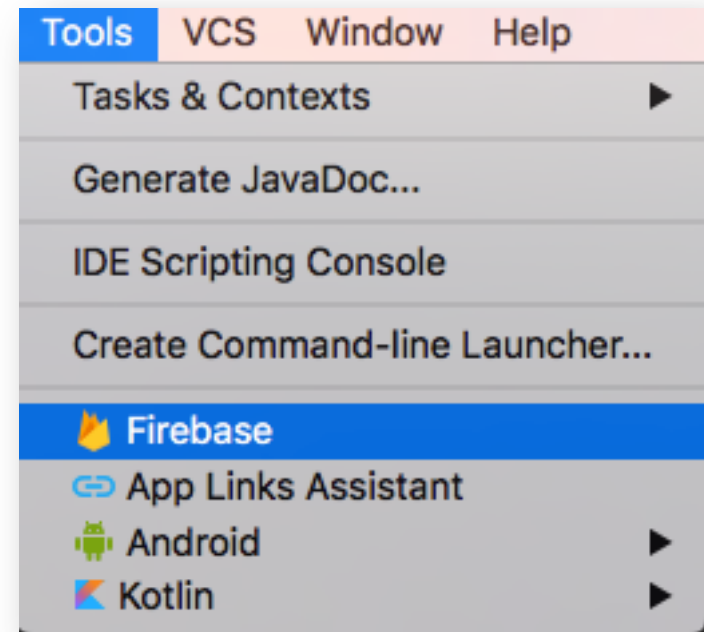
- **Precio del servicio:** gratuito en primera instancia, y escalable a medida que el proyecto crece:
 - <https://firebase.google.com/pricing/>
- Para crear un prototipo de aplicación o una primera versión, el plan gratuito es más que suficiente.

FIREBASE

- Firebase proporciona SDKs (Software Development Kits) para realizar de forma rápida y cómoda toda la comunicación con la base de datos en la nube.
- **Requisitos mínimos del dispositivo**
 - Android 4.0+
 - Google Play Services 11.8.0

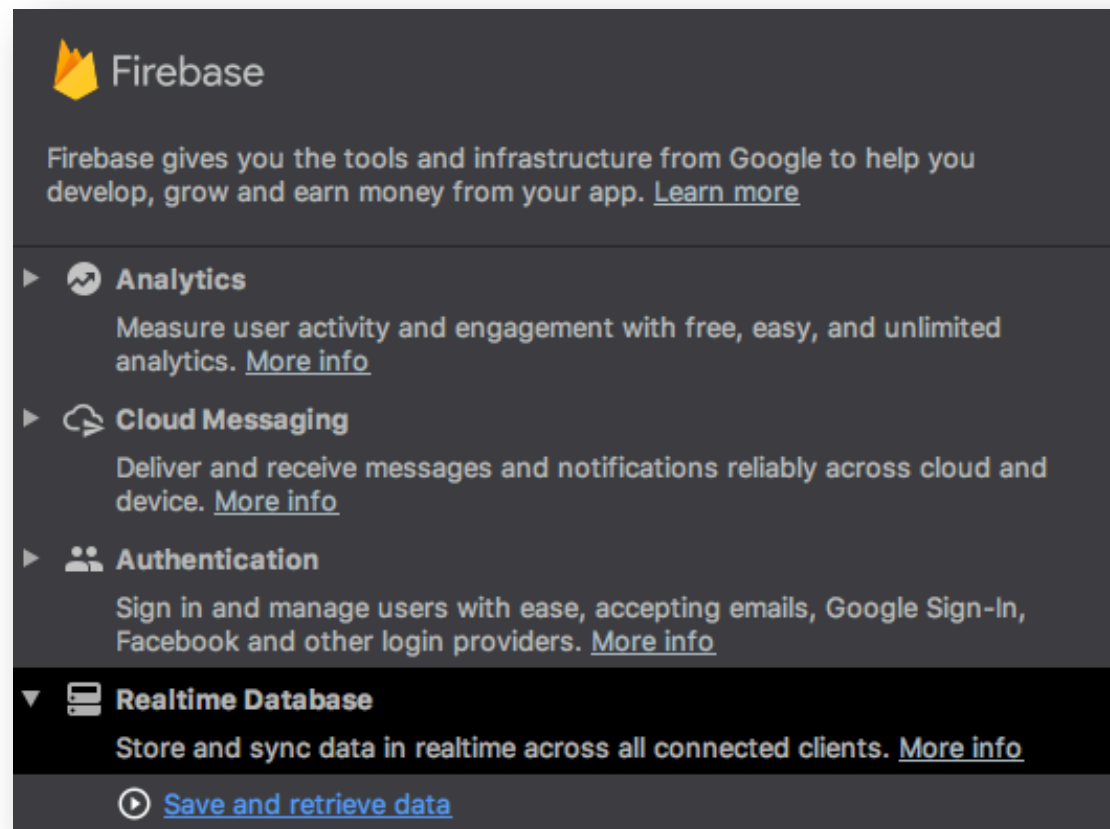
FIREBASE

- Para añadir el SDK de Android a nuestro proyecto, utilizaremos **Firestore Assistant**.
 - Se encuentra en **Herramientas > Firestore** de Android Studio



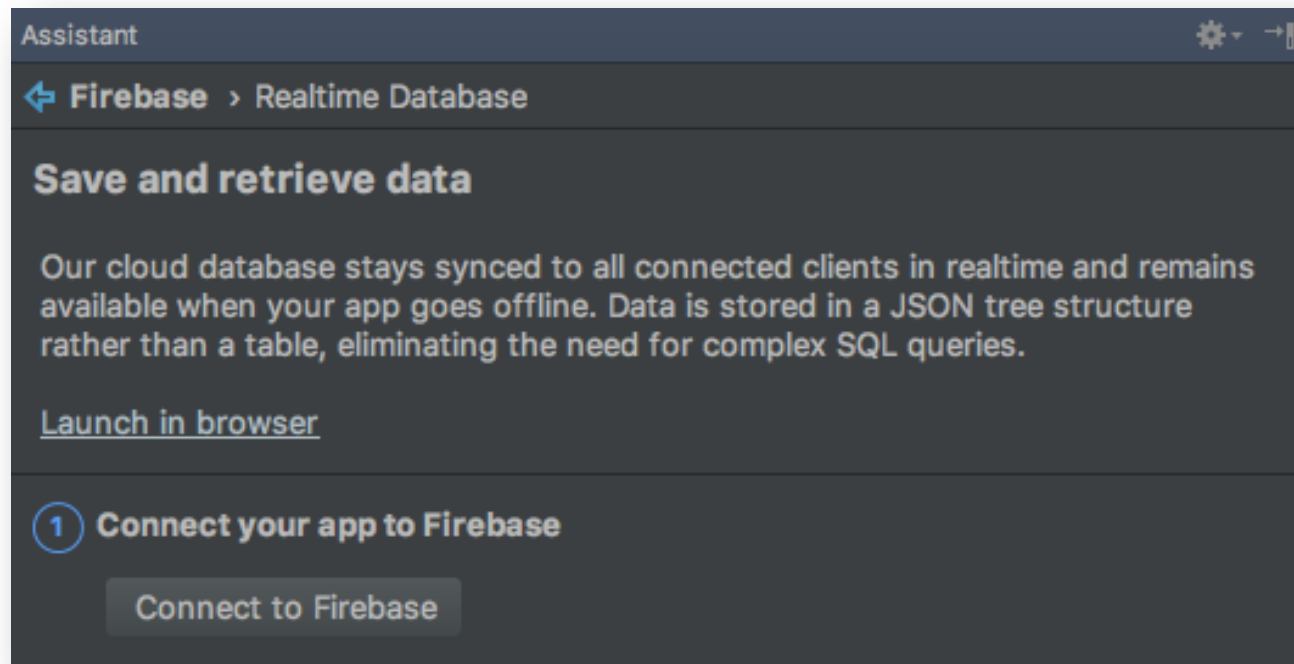
FIREBASE

- Seleccionamos la sección **Realtime Database** y pulsamos sobre ***“Save and Retrieve data”***




FIREBASE

- Acto seguido, debemos **conectarnos con Firebase...**



FIREBASE

Connect to Firebase

 **Firebase**

☒ Create new Firebase project [What's this?](#)


Signed in as **manelizzard@gmail.com** [Sign out](#)

☐ Choose an existing Firebase or Google project

Firebase Demo Project	1 Android app(s) connected, 1 iOS app(s) connected
Follywood	2 Android app(s) connected
Mi Primera App Android	

Country/region [What's this?](#)

By default, your Firebase Analytics data will enhance other Firebase features and Google products. You can control how your Firebase Analytics data is shared in your settings at anytime. [Learn more](#)

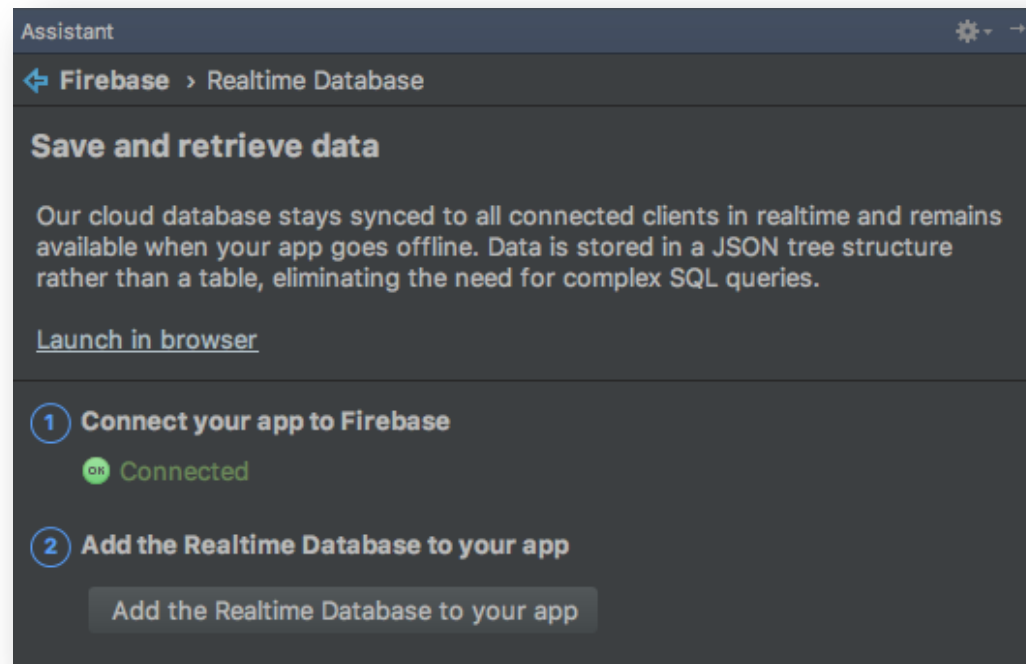


Cancel

Connect to Firebase

FIREBASE

- ... y añadir nuestra base de datos con **“Add the Realtime Database to your app”**



FIREBASE

- **Realtime Database**

- Base de datos **NoSQL** alojada en la nube
- Descrita en formato **JSON**
- Se sincroniza en tiempo real con cada cliente conectado y habilita los datos cuando el dispositivo está *offline*

doc: <https://firebase.google.com/docs/database/?authuser=0>

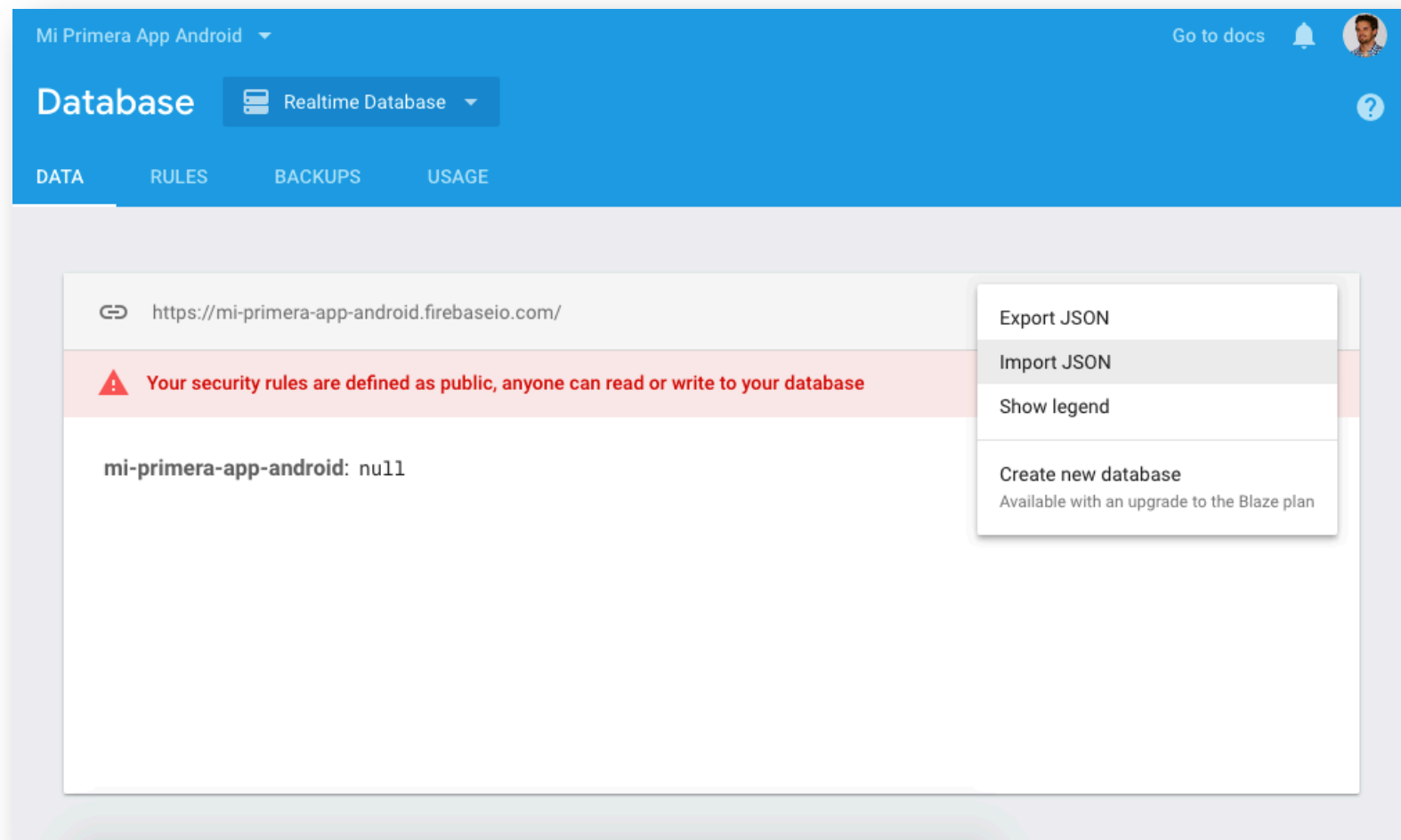
FIREBASE

- **Realtime Database**

- Al crear la base de datos, ésta está vacía por defecto.
- Podemos introducir datos desde la propia consola de Firebase.
- Podemos **importar** una base de datos con un JSON ya existente:
 - https://drive.google.com/open?id=1NrFj5dj9JANp_-9hV2INz6-45GrZ7A_O

FIREBASE

- Importar base de datos en formato JSON



FIREBASE

- Tras crear nuestra primera base de datos en Firebase, podemos realizar peticiones HTTP a la misma.
- **Por ejemplo:**
 - GET <https://mi-primer-app-android.firebaseio.com/games.json>
 - GET <https://mi-primer-app-android.firebaseio.com/games/0.json>

doc: <https://firebase.google.com/docs/reference/rest/database/>

FIREBASE

- Peticiones HTTP básicas en Android
 - Existen varias librerías y utilidades para realizar operaciones HTTP en Android, como *Volley*, *OkHttp*, etc...
 - La forma nativa es utilizar un objeto llamado **URLConnection**. ¡Siempre de forma asíncrona!

FIREBASE

- Integrando **Volley** en nuestra app
 - Volley es una librería para realizar peticiones HTTP de forma sencilla y asíncrona. Google nos recomienda utilizar dicha librería.
 - Para integrarla en el proyecto, modificar **app/build.gradle**

```
dependencies {  
    ...  
    compile 'com.android.volley:volley:1.1.0'  
}
```

FIREBASE

- Integrando **Volley** en nuestra app
 - Una vez integrada la librería, podemos realizar simples peticiones GET, POST, PUT, etc...
 - Las peticiones se construyen con **StringRequest** y se añaden a una cola de peticiones de tipo **RequestQueue**
 - Cada petición define un **Response.Listener** para escuchar la respuesta del servidor

FIREBASE

- Operaciones con **Firestore**:
 - Escribir en nuestra base de datos
 - Leer de nuestra base de datos
 - Actualizar nuestra base de datos
 - Borrar elementos de la base de datos
 - Etc...

FIREBASE

- Escribir en la base de datos de Firebase:
 - La instancia de la base de datos se obtiene con **FirebaseDatabase.getInstance()**

```
// Write a message to the database
FirebaseDatabase database = FirebaseDatabase.getInstance();
DatabaseReference myRef = database.getReference("message");

myRef.setValue("Hello, World!");
```

FIREBASE

- Escribir en la base de datos de Firebase:
 - Se puede escribir cualquier objeto, incluso un objeto Java!

```
// Write a message to the database
FirebaseDatabase database = FirebaseDatabase.getInstance();
DatabaseReference myRef = database.getReference("games");

myRef.setValue(new GameModel(...));
```

FIREBASE

- Leer de nuestra base de datos de Firebase
 - Existen dos métodos principales:
 - **addValueEventListener(...)** – Crea un **listener** que siempre está escuchando cambios en tiempo real en la base de datos
 - **addListenerForSingleValueEvent(...)** – Crea un **listener** que únicamente se ejecuta la primera vez que realizamos la petición, pero no escucha cambios en tiempo real.

FIREBASE

- Leer de nuestra base de datos de Firebase
 - **ValueEventListener** es el objeto que se llama cuando el servidor nos devuelve una respuesta.
 - **Contiene dos métodos:**
 - **onDataChange(DataSnapshot dataSnapshot)**
 - **onCancelled(DatabaseError databaseError)**
 - El primero contiene los datos que hemos pedido a Firebase, mientras el segundo contiene algún tipo de error en caso que suceda

FIREBASE

- Ejemplo:

```
FirebaseDatabase database = FirebaseDatabase.getInstance();
DatabaseReference gamesDatabaseReference =
    database.getReference("games");
gamesDatabaseReference.addListenerForSingleValueEvent(new
 ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        // Aquí recibimos los datos de la base de datos
    }

    @Override
    public void onCancelled(DatabaseError databaseError) {
        // Algún error ha ocurrido
    }
});
```

FIREBASE

- ¿Cómo convertir un objeto de tipo **DataSnapshot** en un objeto de nuestro modelo?
 - Si, por ejemplo, el valor que esperamos recibir es un simple **String**:

```
@Override  
public void onDataChange(DataSnapshot dataSnapshot) {  
    String value = dataSnapshot.getValue(String.class);  
}
```

FIREBASE

- ¿Cómo convertir un objeto de tipo **DataSnapshot** en un objeto de nuestro modelo?
 - Si, por el contrario, el valor que esperamos recibir es un objeto complejo en Java:

```
@Override  
public void onDataChange(DataSnapshot dataSnapshot) {  
    GameModel value = dataSnapshot.getValue(GameModel.class);  
}
```

FIREBASE

- En el caso de recibir un objeto complejo, éste necesita:
 - **Un constructor vacío.**
 - **Getters & Setters.** Los atributos se rellenarán según los nombres de los valores en la base de datos.

```
@Override  
public void onDataChange(DataSnapshot dataSnapshot) {  
    GameModel value = dataSnapshot.getValue(GameModel.class);  
}
```


FIREBASE

- En el caso de recibir una lista de objetos, se puede recorrer dicha lista con **`dataSnapshot.getChildren()`**

```
@Override
public void onDataChange(DataSnapshot dataSnapshot) {
    // Aquí recibimos los datos de la base de datos
    for (DataSnapshot gameSnapthot:dataSnapshot.getChildren())
    {
        GameModel game=gameSnapthot.getValue(GameModel.class);
    }
}
```

PRÁCTICA

1. Adaptando el modelo. Cambiar en **GameModel** los atributos:
 - **iconDrawable** a **icon**, de tipo **String**
 - **backgroundDrawable** a **background**, de tipo **String**
2. Obtendremos errores de compilación al intentar asignar las imágenes tanto en **ListActivity** como en **GameDetailFragment**. Comentar las líneas con errores.

PRÁCTICA

1. Crear una **interfaz Java** llamada **GamesInteractorCallback** con un único método:
 - **void onGamesAvailable();**

```
public interface GamesInteractorCallback {  
    void onGamesAvailable();  
}
```

PRÁCTICA

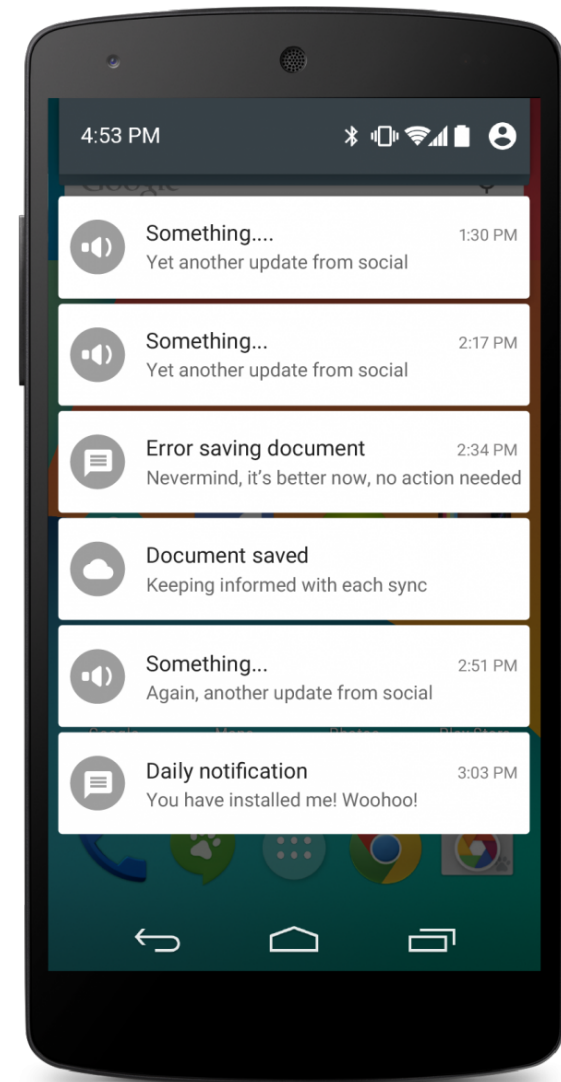
1. Crear una copia de la clase **GamesInteractor** con los siguientes métodos:
 - Constructor vacío
 - **void getGames(GamesInteractorCallback callback)** – Obtendrá el listado de juegos de Firebase y llamará a **callback.onGamesAvailable()** una vez obtenidos y guardados en un ArrayList
 - Podéis llamar a la clase **GamesFirebaseInteractor**
2. Imprimir en **Log.d** los juegos obtenidos desde Firebase

PRÁCTICA

1. En los métodos **onStart** de **ListActivity** y **GameDetailActivity**, invocar **GamesFirebaseInteractor.getGames(new ...)**
2. Cuando se ejecute el método **onGamesAvailable** de la interfaz que pasamos por parámetro, crear los correspondientes *adapters* obteniendo la lista de juegos de **GamesFirebaseInteractor**.

NOTIFICACIONES PUSH

- **Firestore Cloud Messaging** nos permite enviar notificaciones a una aplicación cliente para notificar de:
 - Ciertos datos han cambiado o están disponibles para sincronizar
 - Ha ocurrido algún evento de interés para el usuario



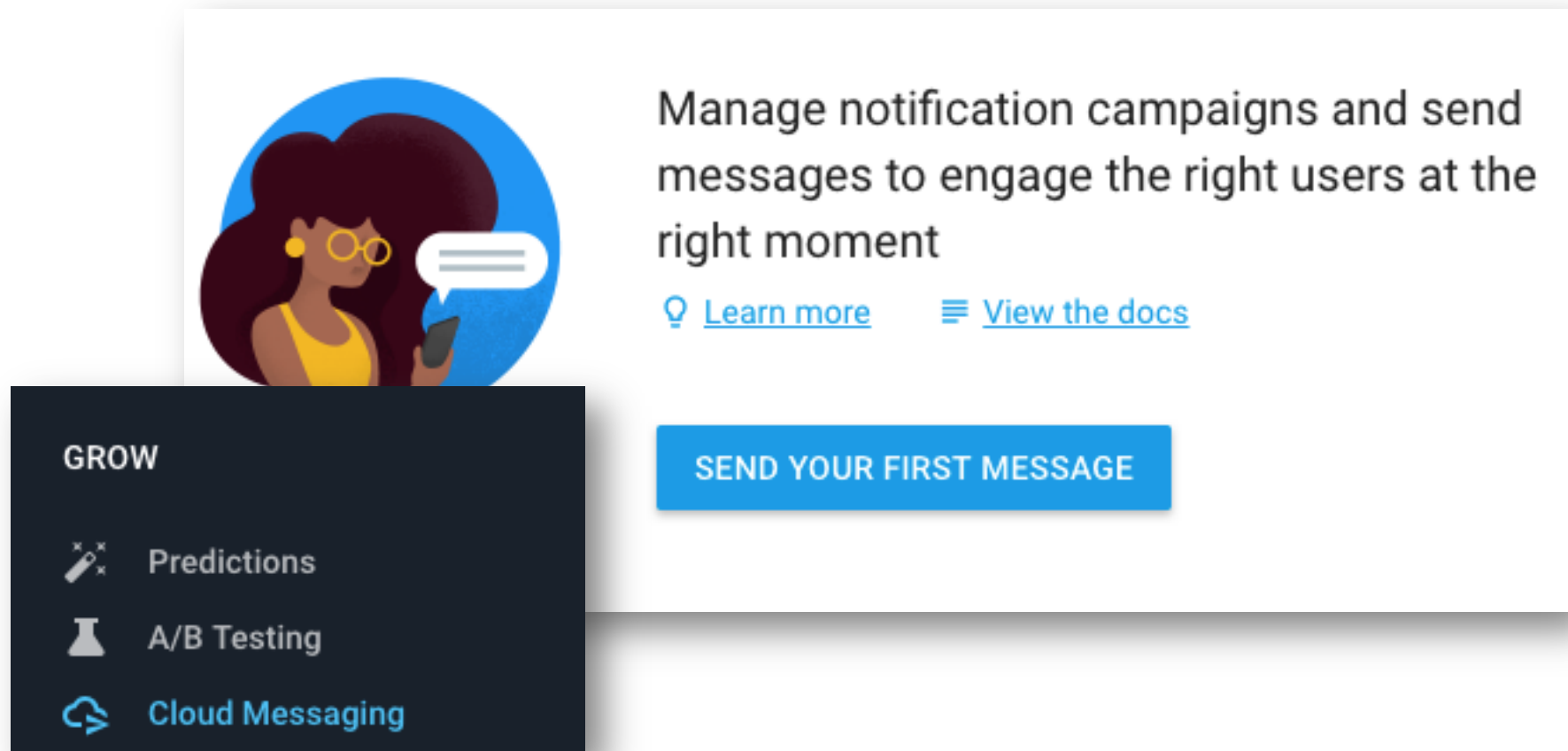
NOTIFICACIONES PUSH

- Para utilizar **Firestore Cloud Messaging**, es necesario importar la librería a través de **app/build.gradle**
 - Sincronizar y ejecutar la aplicación tras el cambio

```
dependencies {  
    ...  
    compile 'com.google.firebase:firebase-messaging:12.0.1'  
}
```

NOTIFICACIONES PUSH

- Desde la consola de Firebase podemos enviar y programar **notificaciones push**



NOTIFICACIONES PUSH

- **Firestore Cloud Messaging** nos ofrece la posibilidad de implementar dos objetos de tipo **Service**:
 - **FirestoreMessagingService** – Nos permite recibir las notificaciones mientras la aplicación está en primer plano, recibir datos o enviar mensajes ascendentes.
 - **FirestoreInstanceIdService** – Nos proporciona un String llamado *Token*, que identifica el dispositivo. Sirve para enviar notificaciones a un único dispositivo (o varios, sabiendo los diferentes tokens)

NOTIFICACIONES PUSH

- **FirebaseMessagingService**
 - Es necesario crear una clase que herede de **FirebaseMessagingService** y declararla en **AndroidManifest.xml**

```
public class MyFirebaseMessagingService
    extends FirebaseMessagingService {

}
```

doc:

<https://firebase.google.com/docs/reference/android/com/google/firebase/messaging/FirebaseMessagingService>

NOTIFICACIONES PUSH

- **FirebaseMessagingService**
 - Es necesario crear una clase que herede de **FirebaseMessagingService** y declararla en **AndroidManifest.xml**

```
<application>
  ...
  <service android:name=". MyFirebaseMessagingService">
    <intent-filter>
      <action
        android:name="com.google.firebase.MESSAGING_EVENT"/>
    </intent-filter>
  </service>
  ...
</application>
```

NOTIFICACIONES PUSH

- Métodos disponibles en **FirebaseMessagingService**
 - **onMessageReceived()** – Se ejecuta cuando se recibe un mensaje.
 - **onMessageSent()** – Cuando se envía un mensaje ascendente
 - **onSendError()** – Cuando hay un error enviando un mensaje ascendente
 - **onDeletedMessages()** – Cuando Firebase borra mensajes pendientes

NOTIFICACIONES PUSH

- Métodos disponibles en **FirebaseMessagingService**
 - **onMessageReceived(RemoteMessage message)**
 - El parámetro recibido contiene datos que pueden resultar útiles para nuestra aplicación. Los datos se obtienen en parejas clave-valor con **message.getData()**

```
@Override
public void onMessageReceived(RemoteMessage remoteMessage) {
    super.onMessageReceived(remoteMessage);
    Map<String, String> data = remoteMessage.getData();
}
```

PRÁCTICA

1. Crear una nueva clase llamada **MyFirebaseMessagingService** que:
 - Herede de **FirebaseMessagingService**
 - Implemente el método **onMessageReceived** y muestre por **Log** los datos que contiene
 - *remoteMessage.getData().toString()*
2. Agregar el componente a **AndroidManifest.xml** tal y como se especifica en las diapositivas anteriores

NOTIFICACIONES PUSH

- **FirebaseInstanceIdService**

- Si necesitamos obtener el token del dispositivo al iniciar la aplicación, podemos hacerlo con
 - **FirebaseInstanceId.getInstance().getToken()**
- Puede devolver **null** si el token aún no existe. Por ello, necesitamos crear una clase que herede de **FirebaseInstanceIdService**

```
String token = FirebaseInstanceId.getInstance().getToken();
```

<https://firebase.google.com/docs/reference/android/com/google/firebase/iid/FirebaseInstanceIdService>

NOTIFICACIONES PUSH

- **FirebaseInstanceIdService**
 - Es necesario crear una clase que herede de **FirebaseInstanceIdService** y declararla en **AndroidManifest.xml**

```
public class MyFirebaseInstanceIdService
    extends FirebaseInstanceIdService {
}
```


NOTIFICACIONES PUSH

- Métodos disponibles en **FirebaseInstanceIdService**
 - **onTokenRefresh()** – Se ejecuta cuando el token ha cambiado. En este momento, debemos actualizar dicho token en el servidor (u otro sitio) para poder enviar notificaciones dirigidas.

```
@Override
public void onTokenRefresh() {
    // Get updated InstanceID token.
    String refreshedToken =
        FirebaseInstanceId.getInstance().getToken();
    Log.d("InstanceIdService", "Refreshed token: " +
        refreshedToken);
}
```

NOTIFICACIONES PUSH

- **FirebaseInstanceIdService**
 - Es necesario crear una clase que herede de **FirebaseInstanceIdService** y declararla en **AndroidManifest.xml**

```
<application>
...
<service android:name=".MyFirebaseInstanceIdService">
    <intent-filter>
        <action
            android:name="com.google.firebase.INSTANCE_ID_EVENT"/>
        </intent-filter>
    </service>
...
</application>
```

PRÁCTICA

1. Al arrancar la aplicación, obtener el **token** de **FirebaseInstanceId** y guardarlo en la base de datos de Firebase

miprimeraapp-db818

```
├── device_push_token: "e-2fsz8mV7g:APA91bEHXuJ1UF5QqIwjQQzz5iYWIuc9Tzq..."  
└── + games
```

2. Crear una nueva clase llamada **MyFirebaseInstanceIdService** que herede de **FirebaseInstanceIdService**
 - Implementar el método **onTokenRefresh**, actualizando el token de la base de datos de Firebase

NOTIFICACIONES PUSH

- Se pueden cambiar tanto el icono como el color del mismo cuando salta una notificación
- Para ello, debemos especificar unos **metadatos** en **AndroidManifest.xml**, dentro del tag **<application>**

```
<meta-data
    android:name="com.google.firebase.messaging.default_notification_icon"
    android:resource="@drawable/ic_stat_ic_notification" />
<meta-data
    android:name="com.google.firebase.messaging.default_notification_color"
    android:resource="@color/colorAccent" />
```

PRÁCTICA

1. Cambiar el icono y color de notificación cuando recibimos un push
 - Obtener un icono de notificación que nos guste de la web **Android Asset Studio** y añadirlo al proyecto
 - Definir un color para la notificación
 - Añadir los **meta** tags en AndroidManifest.xml
2. Probar que realmente funciona enviando una notificación push desde Firebase

PRÁCTICA

