

Desarrollo de apps con Android



12. Almacenamiento de datos

Almacenamiento en Android (1/2)

Se consideran distintos tipos de [almacenamiento de datos en Android](#):

- **Almacenamiento específico de la app:** Ficheros que solamente va a utilizar la app haciendo uso del almacenamiento interno o externo del dispositivo. Utilizar los directorios del almacenamiento interno para guardar **la información sensible a la que no se debería acceder desde otras apps.**
- **Almacenamiento compartido:** Ficheros que se pueden compartir con otras apps como podrían ser archivos multimedia y ciertos documentos, tanto en la memoria interna del dispositivo como en medios de almacenamiento externos, como sería una tarjeta SD.
- **Preferencias:** Datos privados y primitivos en forma de pares *clave-valor*.

Almacenamiento en Android (2/2)

- **Bases de datos:** Almacenamiento de datos estructurados en una base de datos privada a través de la librería [Room](#).
- [Proveedores de contenido](#): Gestión del acceso a los datos que las diferentes apps almacenan proporcionando una forma de compartir datos entre ellas. Encapsulan los datos y proporcionan mecanismos para definir su seguridad. Ejemplos: contactos, ficheros multimedia y el historial de navegación.
- **Internet:** Ficheros que se almacenan y se recuperan de la nube.
Ejemplo: [Firebase Realtime Database](#) ➡ [Conexión con app de Android](#).

Ficheros

Sistema de archivos de Android (1/2)

- **Almacenamiento interno:** Datos privados en la memoria del dispositivo. Disponible en todos los dispositivos Android. Adecuado para almacenar los datos de los que depende la app.
- **Almacenamiento externo:** Datos públicos en el propio dispositivo Android o en almacenamiento externo.
 - En la mayoría de los dispositivos, el almacenamiento externo es mayor al interno.
 - Los componentes externos al dispositivo, como las tarjetas SD, aparecen en el sistema de archivos como parte del almacenamiento externo.
- La localización exacta de ubicación de los ficheros de la app varía en dispositivos distintos. No meter las rutas de los ficheros de manera manual ("hard-coded").

Sistema de archivos de Android (2/2)

- Las apps pueden mostrar la estructura de directorios.
- La estructura y las operaciones son similares a Linux y java.io.
- Para ver los archivos almacenados a un dispositivo hacer uso del Explorador de archivos del dispositivo disponible en Android Studio ([Device Explorer](#)).

Almacenamiento interno

- Siempre disponible.
- Utiliza el sistema de archivos del dispositivo.
- Solamente la propia app puede acceder a los ficheros, a no ser que se dé permisos para que se puedan leer o escribir.
- Al desinstalar la app, el sistema Android borrará todos los ficheros del almacenamiento interno.

Almacenamiento externo

- No están siempre disponibles, ya que se pueden desconectar o extraer del dispositivo.
- Utiliza el sistema de ficheros del dispositivo o almacenamiento externo como las tarjetas SD.
- Cualquier app puede leer los datos.
- Al desinstalar la app, el sistema no elimina los ficheros que son privados de la app.

Almacenamiento interno vs externo

- El almacenamiento interno será la mejor opción:
 - Cuando no se desea que otras apps accedan a los ficheros de la app.
 - Ficheros temporales (inferiores a 1MB).
- El almacenamiento externo será la mejor opción:
 - No se tienen restricciones de acceso.
 - Se desea compartir la información con otras apps.
 - Se permite al usuario el acceso a los datos desde un ordenador.
 - Ficheros temporales (superiores a 1MB).

Ficheros del usuario

- Guardar en un directorio público los ficheros del usuario que ha obtenido a través de la app, para que así otras apps puedan acceder también, y para que el usuario tenga opción de copiarlos desde el dispositivo.
- Guardar los ficheros externos en directorios públicos.
 - `android.os.Environment` define constantes para [directorios públicos estándares](#) que todas las apps pueden utilizar (ej., `Environment.DIRECTORY_RINGTONES`).

Almacenamiento interno

Almacenamiento interno (1/2)

- Utilizar los directorios privados solamente para la app.
- La app siempre tiene permisos para leer/escribir.
- Directorio de almacenamiento permanente: [getFilesDir\(\)](#)
- Directorio de almacenamiento temporal: [getCacheDir\(\)](#)
- Crear/Abrir directorio en almacenamiento interno: [getDir\(\)](#)

Almacenamiento interno (2/2)

- Las apps por sí mismas se almacenan por defecto haciendo uso del almacenamiento interno. Si el tamaño del APK es demasiado grande se puede indicar una preferencia en el fichero `AndroidManifest.xml` de la app:

```
<manifest ...  
    android:installLocation="preferExternal">  
    ...  
</manifest>
```

Almacenamiento interno: Crear un fichero

```
File file = new File(context.getFilesDir(), filename);
```

Utilizar los operadores o streams de gestión de ficheros del estándar [java.io](#) para interactuar con los ficheros.

Almacenamiento interno: Gestión ficheros (1/4)

Se han añadido métodos adicionales a `Context` para la gestión de ficheros de memoria interna:

- Método [`openFileInput\(\)`](#): abre un fichero para lectura.
- Método [`openFileOutput\(\)`](#): abre un fichero para escritura.
- El nombre del fichero no puede contener subdirectorios.
- Siempre hay que cerrar los ficheros con el método `close()`.
- Prestar atención a la gestión de errores (`try/catch`).

Almacenamiento interno: Gestión ficheros (2/4)

```
public class RepositoryPointsInternal {  
    ...  
    public RepositoryPointsInternal (Context context){  
        this.context = context;  
    }  
    public void savePoints(int points, String name, long game_date){  
        try {  
            FileOutputStream f = context.openFileOutput(POINTS_FILE,  
                                                         Context.MODE_APPEND);  
            String fileText = points + " " + name + " " + game_date + "\n";  
            f.write(fileText.getBytes());  
            f.close();  
        } catch (java.io.IOException e) {  
            Log.e("MyApp", e.getMessage(), e);  
        }  
    }  
    ...  
}
```



Sigue

Almacenamiento interno: Gestión ficheros (3/4)

```
...
    public List<String> showPoints(){
        try {
            List<String> result = new ArrayList<String>();
            FileInputStream f = context.openFileInput(POINTS_FILE);
            BufferedReader input = new BufferedReader(new InputStreamReader(f));
            int n = 0;
            String line;
            do {
                line = input.readLine();
                if (line != null) {
                    result.add(line);
                    n++;
                }
            } while (n < 10 && line != null);
            f.close();
            return result;
        } catch (java.io.IOException e) {
            Log.e("MyApp", e.getMessage(), e);
        }
    }
...
}
```

Almacenamiento interno: Gestión ficheros (4/4)

- Ejemplo: MainActivity.java ➡ método onCreate():

```
RepositoryPointsInternal internal_repository =  
    new RepositoryPointsInternal(this);
```

Almacenamiento externo

Almacenamiento externo

- Dispositivo (no extraíble) o tarjeta SD.
- Requiere establecer permisos en `AndroidManifest.xml`.
 - El permiso de escritura incluye el de lectura.

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

Asegurar disponibilidad del almacenamiento

Hay que asegurarse de que el almacenamiento externo está disponible, ya que, por ejemplo, se podría haber extraído la tarjeta SD:

```
public boolean isExternalStorageWritable() {  
    String state = Environment.getExternalStorageState();  
    if (Environment.MEDIA_MOUNTED.equals(state)) {  
        //Se puede leer y escribir.  
        return true;  
    }  
    return false;  
}
```

Ejemplos de directorios externos públicos

- Ficheros de audio que el usuario puede utilizar para alarmas y tonos de llamada:

`Environment.DIRECTORY ALARMS` y
`Environment.DIRECTORY RINGTONES`

- Documentos que ha sido creados por el usuario:

`Environment.DIRECTORY DOCUMENTS`

- Ficheros descargados por el usuario:

`Environment.DIRECTORY DOWNLOADS`

Acceso a directorios externos públicos

1. Obtener la ruta a través de

[Environment.getExternalStoragePublicDirectory\(\)](#)

2. Crear fichero:

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

```
File file = new File(path, "DemoPicture.jpg");
```


Acceso a directorios externos privados

1. Obtener la ruta a través de [`getExternalFilesDir\(\)`](#). Pasar un String para indicar el tipo de directorio que se desea devolver (`Environment.DIRECTORY_MUSIC`, `Environment.DIRECTORY_PODCASTS...`, o null para el directorio raíz).
2. Crear el fichero:

```
File path = getExternalFilesDir(null);  
File file = new File(path, "DemoPicture.jpg");
```

Espacio disponible de almacenamiento

- Si Android no tiene espacio suficiente lanza la excepción [IOException](#).
- Si no se conoce el tamaño del fichero, comprobar el espacio disponible utilizando métodos del paquete [java.io.File](#):
 - [getFreeSpace\(\)](#)
 - [getTotalSpace\(\)](#)
- Si se desconoce el espacio que se va a necesitar:
 - try/catch [IOException](#).

Borrado de ficheros

- Almacenamiento externo:

```
myFile.delete();
```

- Almacenamiento interno:

```
myContext.deleteFile(fileName);
```

- Periódicamente se deberían eliminar todos los ficheros que han sido generados temporalmente a través del método `getCacheDir()`.

Respetar los ficheros del usuario

- Cuando se desinstala una app, se elimina todo el directorio de almacenamiento privado de la app con su contenido.
- Por tanto, **no se debe utilizar el almacenamiento privado para los contenidos que realmente pertenecen al usuario**. Por ejemplo:
 - Fotos capturadas o editadas por la app.
 - Música que el usuario haya podido adquirir a través de la app.


Preferencias

¿Qué son las preferencias?

- Representan la configuración de la app.
- Permite leer y escribir pequeñas cantidades de datos privados de la app y primitivos en forma de pares **clave-valor** que se guardan en un fichero haciendo uso del almacenamiento del dispositivo.
- [SharedPreferences](#): Interfaz que permite acceder y modificar los datos de preferencias devuelto por el método [getSharedPreferences\(String, int\)](#).
- Utilizar el ciclo de vida de la `Activity` para grabar la información:
 - Guardar los datos en el método **onPause()**
 - Recuperar los datos en el método **onCreate()**

Shared Preferences vs. Saved Instance State

Tema 2 :: Guardar y recuperar estado Activity

Shared Preferences	Saved Instance State 
Se mantiene entre distintas sesiones, da igual que la app se pare o se reinicie, e incluso si el dispositivo se reinicia.	Conserva los datos del estado a través de las instancias de la <code>Activity</code> , pero solamente en la misma sesión de usuario.
Utilizado para datos que se deben recordar en las distintas sesiones del usuario, como sería, por ejemplo, una configuración específica o la puntuación obtenida en un juego.	Utilizado para datos que no requieren ser recordados entre distintas sesiones del usuario, como sería, por ejemplo, la opción de menú que tenía seleccionada el usuario, o el estado actual de una <code>Activity</code> .
Representado por un número pequeño de pares clave-valor.	Representado por un número pequeño de pares clave-valor.
Los datos son privados para la app.	Los datos son privados para la app.
Se suele utilizar para almacenar las preferencias del usuario.	Se suele utilizar para recrear el estado una vez que el dispositivo ha cambiado de orientación.

Creación de preferencias

- Solamente se requiere un fichero de preferencias por app.
- Nombrarlo con el paquete de la app, pues así será único y fácil de asociar a la app que le corresponda.
- El argumento MODE (int) que se utiliza en el método `getSharedPreferences(String, int)` se da por compatibilidad hacia atrás: utilizar solamente la opción `MODE_PRIVATE` para garantizar así la seguridad en la app.

getSharedPreferences()

```
package es.ucm.fdi.helloworld;

import android.content.SharedPreferences;

...

public class MainActivity extends AppCompatActivity {
    private static final String sharedPrefFile =
        "es.ucm.fdi.helloworldsharedprefs";
    private SharedPreferences mPreferences;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
    }

    ...
}
```

Almacenamiento de preferencias (1/2)

- Guardar las preferencias en el método del evento `onPause()` del ciclo de vida de la `Activity`.
- [SharedPreferences.Editor](#): Interfaz que se utiliza para modificar los valores del objeto [SharedPreferences](#).
- Todos los cambios realizados se quedan pendientes y no se copian al original `SharedPreferences` hasta que no se invoque el método [commit\(\)](#) o el método [apply\(\)](#).

Almacenamiento de preferencias (2/2)

- Métodos **put*** (`putString(String key, String value)`, `putBoolean(String key, boolean value)`...) sobrescribe si la clave existe.
- `apply()` guarda de manera segura y asíncrona (es decir, aparte del hilo UI).
- `commit()` guarda la preferencias en un almacenamiento persistente de manera síncrona. Si no interesa el valor de retorno y se está invocando desde el hilo principal de la app, se debería utilizar mejor el método `apply()`.

SharedPreferences.Editor

```
@Override
protected void onPause() {
    super.onPause();
    mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
    SharedPreferences.Editor preferencesEditor =
        mPreferences.edit();
    preferencesEditor.putInt("count", mCount);
    preferencesEditor.putInt("color", mCurrentColor);
    preferencesEditor.apply();
}
```

Recuperación de las preferencias (1/2)

- Recuperar en el método del evento `onCreate()` del ciclo de vida de la `Activity`.
- Los métodos `Get` consideran dos argumentos: la clave, y el valor por defecto para el caso de que la clave no se encuentre entre las preferencias.
- Al utilizar el valor por defecto se elimina la necesidad de estar comprobando si existe la preferencia en el fichero.

Recuperación de las preferencias (2/2)

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
    mCount = mPreferences.getInt("count", 1);
    mShowCount.setText(String.format("%s", mCount));
    mCurrentColor = mPreferences.getInt("color", mCurrentColor);
    mShowCount.setBackgroundColor(mCurrentColor);
    mNewText = mPreferences.getString("text", mNewText);
    ...
}
```

Borrado de preferencias

- Llamar al método `clear()` de `SharedPreferences.Editor` y aplicar los cambios.
- Se pueden combinar llamadas a `put()` y `clear()`. Sin embargo, cuando se ejecuta el método `apply()` lo primero que se va a ejecutar es el método `clear()`.

clear()

```
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);  
SharedPreferences.Editor preferencesEditor =  
    mPreferences.edit();  
preferencesEditor.clear();  
preferencesEditor.apply();
```


Seguimiento de cambios en preferencias

Seguimiento de cambios

Para recibir notificación cada vez que se modifiquen las preferencias de usuario (ej., a través de una página de configuración en la propia app):

- Crear el *listener* que sobrescriba el método del evento **`onSharedPreferencesChanged()`**.
- Registrar *listener* con [`registerOnSharedPreferencesChangeListener\(\)`](#).
- Registrar o eliminar registro en los métodos de los eventos [`onResume\(\)`](#) y [`onPause\(\)`](#).

Creación y registro del listener

```
SharedPreferences.OnSharedPreferenceChangeListener listener =  
    new SharedPreferences.OnSharedPreferenceChangeListener() {  
        @Override  
        public void onSharedPreferenceChanged(  
            SharedPreferences prefs, String key) {  
            // Implementación del listener  
            if (key.equals("count")) {  
                // Implementación para la clave "count".  
            }  
        }  
    }  
};  
  
@Override protected void onResume() {  
    super.onResume();  
    mPreferences = getSharedPreferences(mSharedPrefFile, MODE_PRIVATE);  
    mPreferences.registerOnSharedPreferenceChangeListener(listener);  
}
```

App settings

¿Qué son las app settings?

- A través de la opción [app settings](#), los usuarios pueden establecer características y comportamiento de la app (es decir, su configuración).
Ejemplos:
 - La dirección (localización) de casa, moneda por defecto.
 - Comportamiento antes notificaciones para una app específica.
- Se utiliza para valores que no suelen cambiar con frecuencia, pero que son relevantes para la mayoría de los usuarios.
- Si los valores cambian frecuentemente, considerar incluirlo como opción de menú o del panel de navegación.

Ejemplos de app settings

Favorite destination

San Francisco

CANCEL OK

Sleep through meals?

You will not be woken for meals

☒

Preferred snack

☐ chocolate

☒ ice cream

☐ fruit

☐ nuts

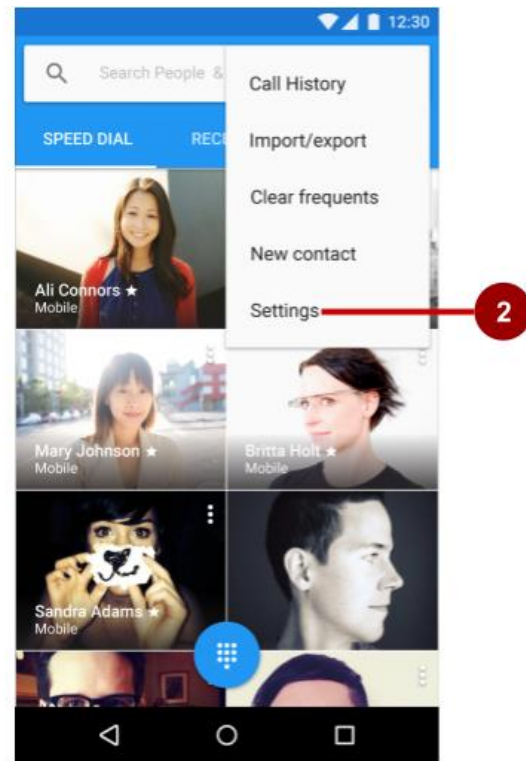
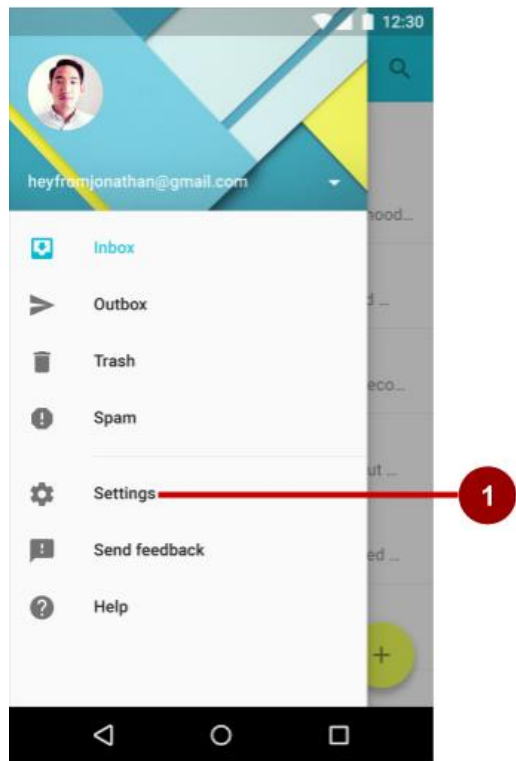
CANCEL

Acceso a ap settings

Los usuarios acceden a las opciones de configuración de la app a través de:

1. Panel de navegación
2. Opciones de menú

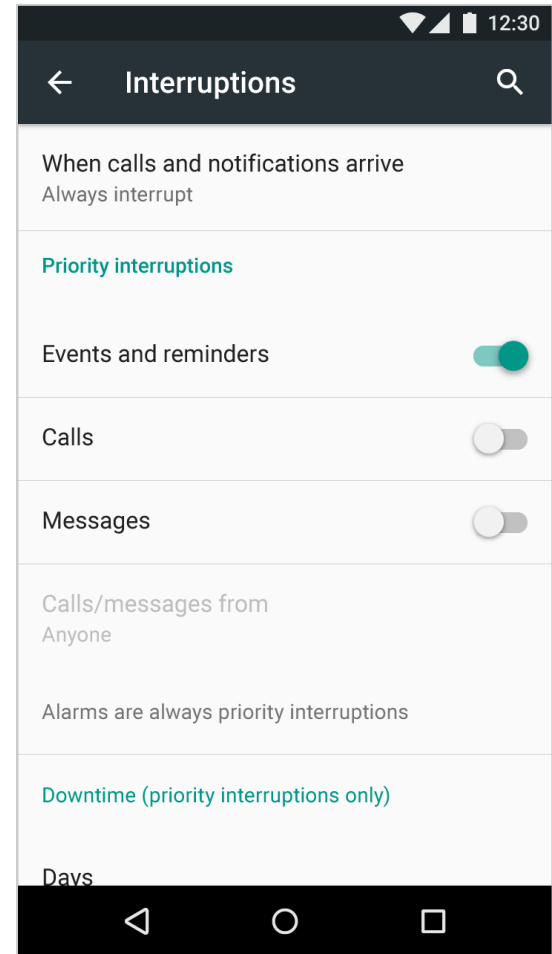
Buenas prácticas en diseño es situar la configuración al final, después del resto de opciones (excepto para la opciones de *Ayuda y Envío de Feedback*).



Setting screens

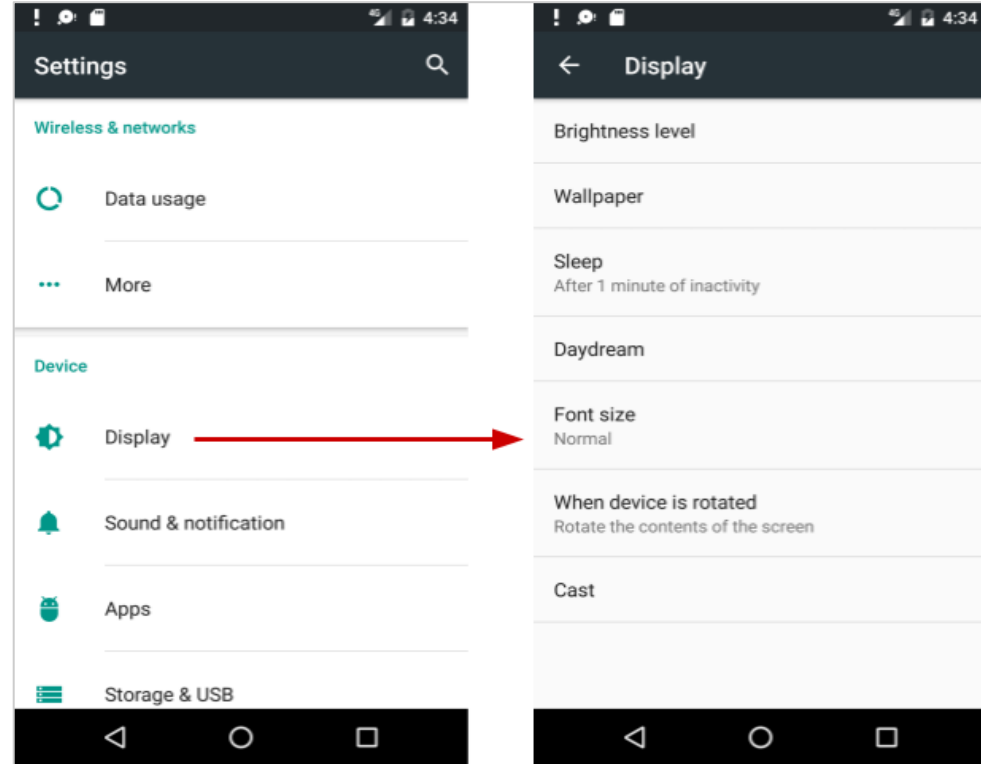
Organizar app settings

- Número manejable de opciones.
- La organización estándar depende del número de elementos de configuración que se requieran:
 - Menos de 7: disponer de acuerdo a su prioridad situando el más importante en primer lugar.
 - 7-15: establecer grupos de elementos de configuración que estén relacionados (se identificarán visualmente a través de una línea divisoria).



+16 elementos de configuración

- Agrupar en ventanas que se muestran a partir de la pantalla principal de Configuración.



Clase Preference

- [Preference](#) proporciona una `View` para cada configuración.
- Asocia la `View` con la interfaz de [SharedPreferences](#) para almacenar y recuperar los datos de configuración.
- Utilizar la clave en la preferencia para poder asociarle el valor del elemento de configuración.
- Almacena pares clave-valor en el fichero por defecto de `SharedPreferences` (librería **androidx.preference:preference**):

[PreferenceManager](#).getDefaultSharedPreferences(this)

Ejemplos de subclases de Preference

- [CheckBoxPreference](#)
- [ListPreference](#)
- [SwitchPreference](#)
- [EditTextPreference](#)
- [Más información en androidx.preference](#)

Clases de agrupación de preferencias

- [PreferenceScreen](#)

- Raíz de la jerarquía layout de preferencias.
- Al comienzo de cada [pantalla de configuración](#).

- [PreferenceCategory](#)

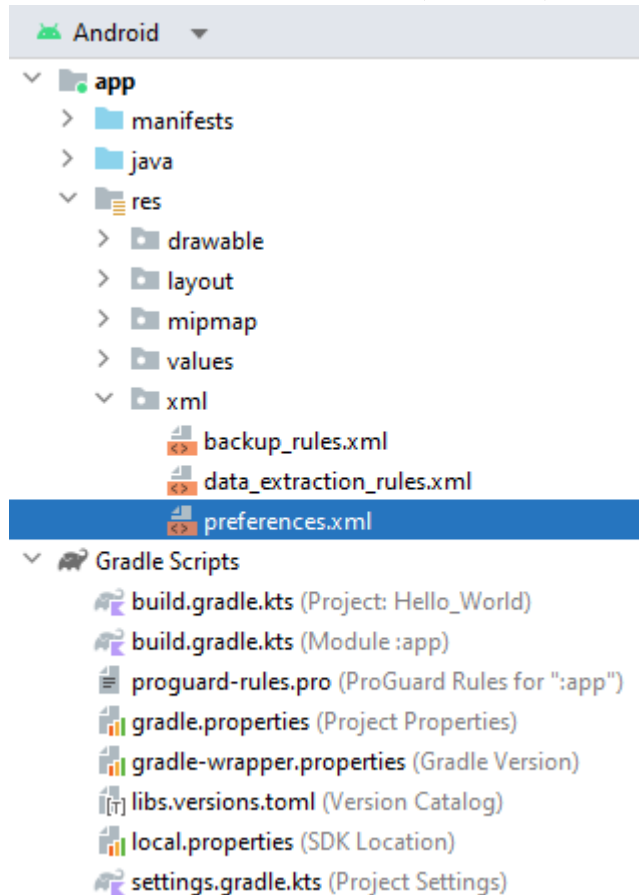
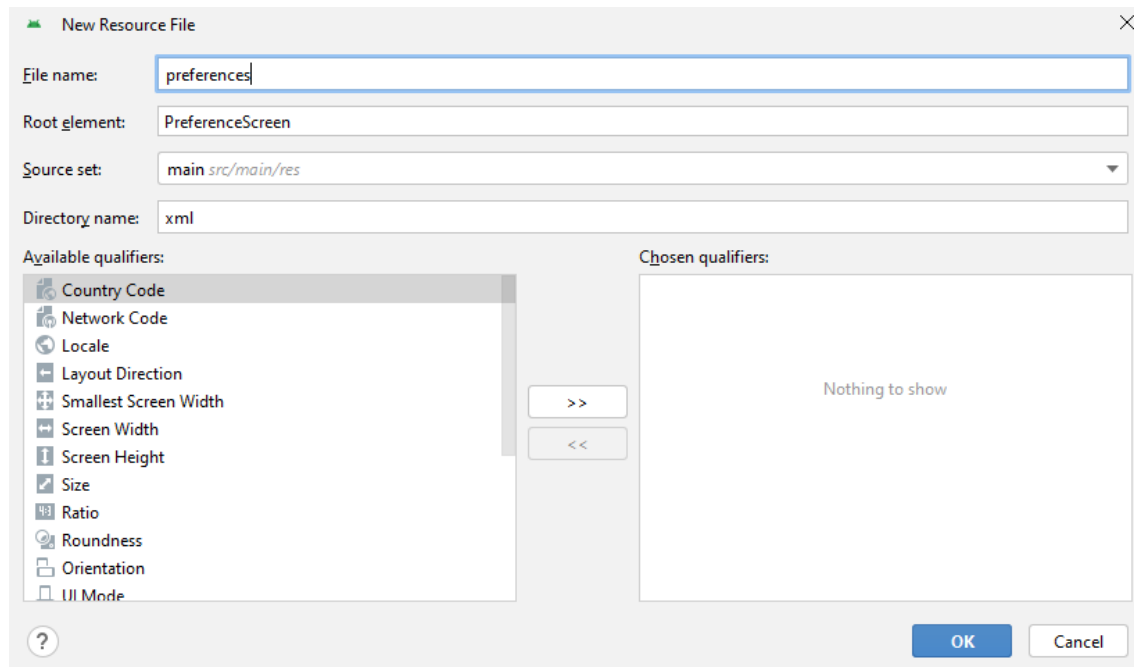
- Título a mostrar sobre un grupo de preferencias como una sección aparte.

Definir Configuración como PreferenceScreen (1/2)

- Establecer la configuración en una ventana de preferencias (`PreferenceScreen`).
- Es como un layout.
- Definirlo a través de un fichero de preferencias (carpeta **xml**):

res > New > Android Resource file > preferences.xml

Definir Configuración como PreferenceScreen (2/2)



Ejemplo Preference Screen

<PreferenceScreen

```
xmlns:app="http://schemas.android.com/apk/res-auto">
```

```
<PreferenceCategory
```

```
app:title="Flight Preferences">
```

```
<CheckBoxPreference
```

```
app:title="Wake for meals"
```

```
... />
```

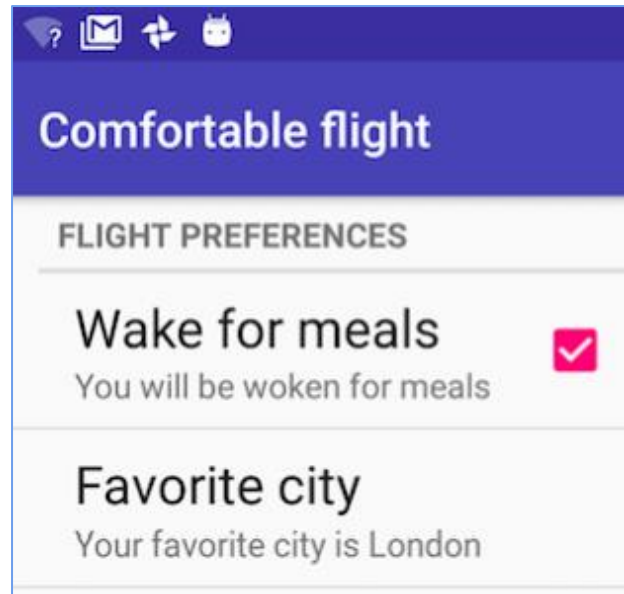
```
<EditTextPreference
```

```
app:title="Favorite city"
```

```
.../>
```

```
</PreferenceCategory>
```

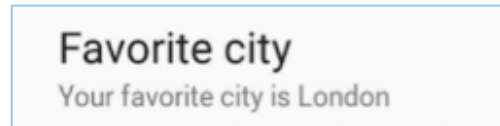
```
</PreferenceScreen>
```



Cada preferencia ha de indicar su clave

- Cada preferencia tendrá su clave asociada.
- Android utiliza la clave para guardar el valor de configuración en el fichero por defecto de *SharedPreferences* vinculado a la app.

```
<EditTextPreference  
    app:title="Favorite city"  
    app:key="fav_city"  
    ... />
```



SwitchPreference

```
<PreferenceScreen
```

```
xmlns:android="http://schemas.android.com/apk/res/android">
```

```
<SwitchPreference
```

```
    android:defaultValue="true"
```

```
    android:title="@string/pref_title_social"
```

```
    android:key="switch"
```

```
    android:summary="@string/pref_sum_social" />
```

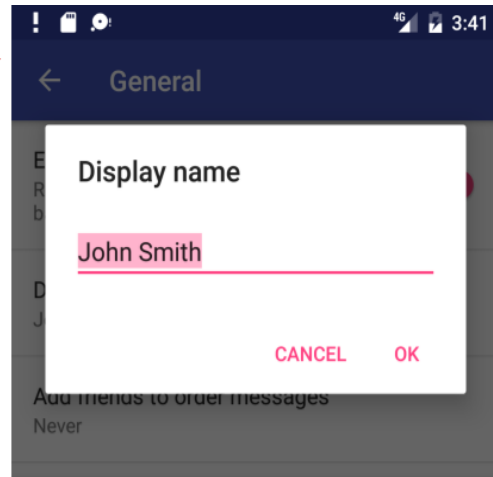
```
</PreferenceScreen >
```

Enable social recommendations

Recommendations for people to contact
based on your order history



EditTextPreference



```
<EditTextPreference
```

```
    android:capitalize="words"
```

```
    android:inputType="textCapWords"
```

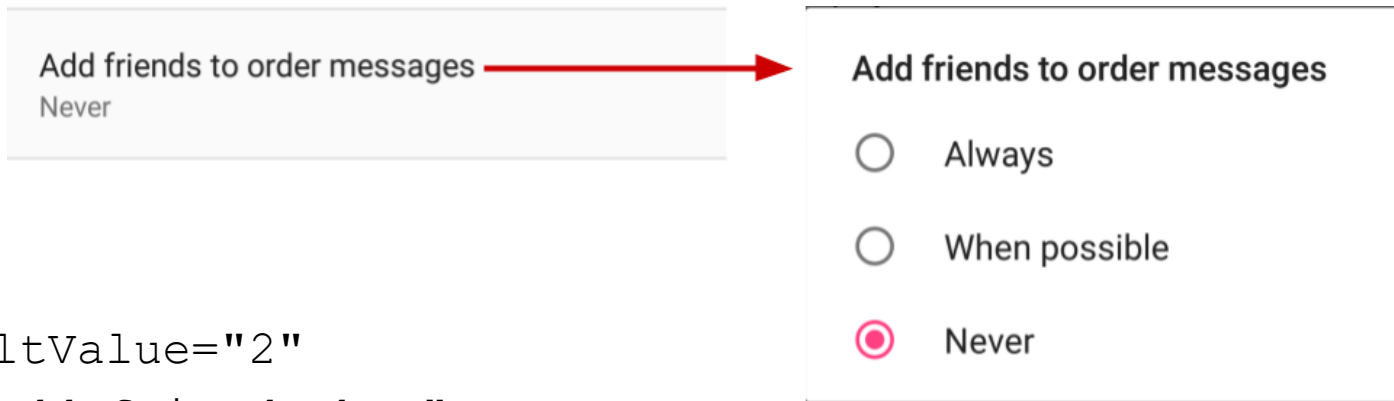
```
    android:key="user_display_name"
```

```
    android:maxLines="1"
```

```
    android:defaultValue="@string/pref_default_display_name"
```

```
    android:title="@string/pref_title_display_name" />
```

ListPreference (1/2)



```
<ListPreference
    android:defaultValue="2"
    android:key="add_friends_key"
    android:entries="@array/pref_example_list_titles"
    android:entryValues="@array/pref_example_list_values"
    android:title="@string/pref_title_add_friends_to_messages"
/>
```

ListPreference (2/2)

- `android:entries`: Array de etiquetas para los radio button.
- `android:entryValues`: Array de valores para los radio button.

Guardar y recuperar app settings

Guardar y recuperar app settings

- No hay que programar nada para que se guarde la configuración.
- Para recuperar las preferencias:

```
SharedPreferences pref =  
    PreferenceManager.getDefaultSharedPreferences(this);  
boolean isActive= pref.getBoolean("checkboxActivePref",true)  
String order = pref.getString("order","?");
```

Base de datos local con Room

Room para persistencia de datos (1/3)

- Almacenamiento local en el dispositivo.
- [Room](#) representa una capa de abstracción sobre [SQLite](#) que permite acceder a la base de datos de una manera sencilla aprovechando toda su tecnología.
- Hace uso de modelo de datos con anotaciones que permiten minimizar el código estándar repetitivo y propenso a errores.
- Añadir la librería en las dependencias de la app (fichero **build.gradle.kts**):

```
val room_version = "2.6.1"
```

```
implementation("androidx.room:room-runtime:$room_version")
```

```
annotationProcessor("androidx.room:room-compiler:$room_version")
```

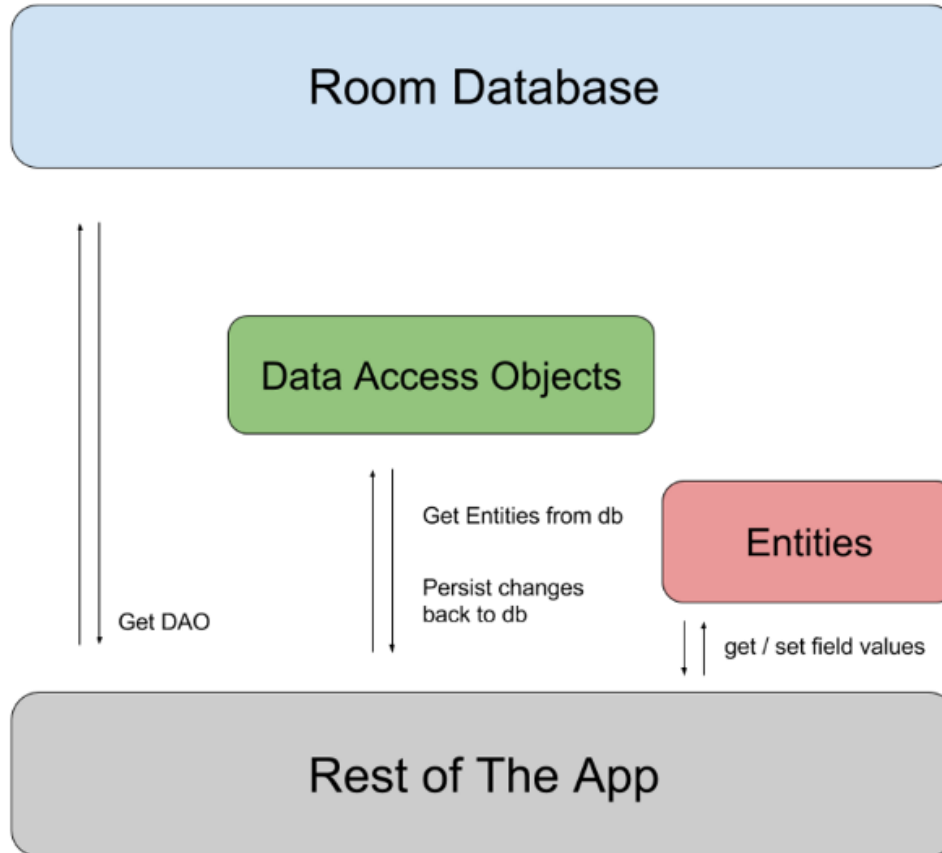
Room para persistencia de datos (2/3)

- **Componentes principales:**

- [Clase de la base de datos](#): contiene la base de datos y sirve como punto de acceso principal para la conexión subyacente a los datos persistentes de la app.
- [Entidades de datos](#): Representan las tablas de la base de datos que utiliza la app.
- [Objetos de acceso a datos \(DAOs\)](#): proporcionan métodos para que la app pueda consultar, actualizar, insertar y borrar datos en la base de datos.

Room para persistencia de datos (3/3)

- **Arquitectura:**



Entidad de datos

```
@Entity
public class User {
    @PrimaryKey
    public int uid;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

Objeto de acceso a datos (DAO)

@Dao

```
public interface UserDao {  
    @Query("SELECT * FROM user")  
    List<User> getAll();  
    @Query("SELECT * FROM user WHERE uid IN (:userIds)")  
    List<User> loadAllByIds(int[] userIds);  
    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +  
            "last_name LIKE :last LIMIT 1")  
    User findByName(String first, String last);  
    @Insert  
    void insertAll(User... users);  
    @Delete  
    void delete(User user);  
}
```

Base de datos

```
@Database(entities = {User.class}, version = 1)

public abstract class AppDatabase extends RoomDatabase {
    // Para cada clase DAO que se asoció con la BD
    // debe definir un método abstracto que tenga cero argumentos
    // y muestre una instancia de la clase DAO.
    public abstract UserDao userDao();

    // Crear una instancia de la base de datos
    public static AppDatabase instance;
    public static AppDatabase getInstance(Context context) {
        if (instance == null) {
            instance = Room.databaseBuilder(context, AppDatabase.class, "myappdb")
                .allowMainThreadQueries()
                .fallbackToDestructiveMigration()
                .build();
        }
        return instance;
    }
}
```

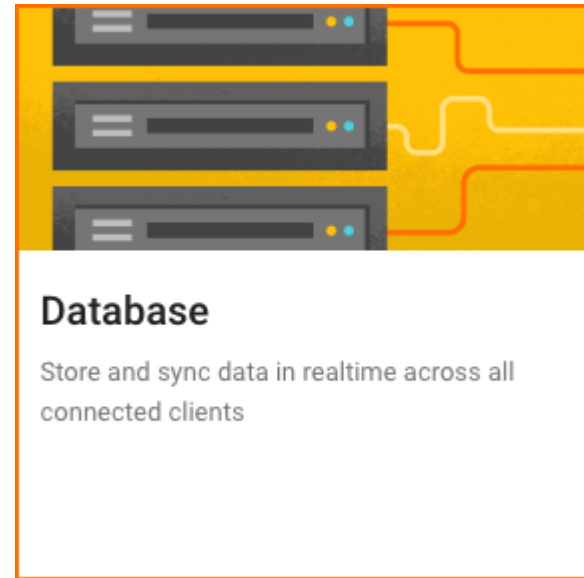
Otras opciones de almacenamiento

Utilizar Firebase para almacenar y compartir datos

Almacenar y sincronizar datos con la base de datos Firebase disponible en la nube.

Los datos se sincronizan para todos los clientes y permanece disponible aunque la app se encuentre offline.

firebase.google.com/docs/database/



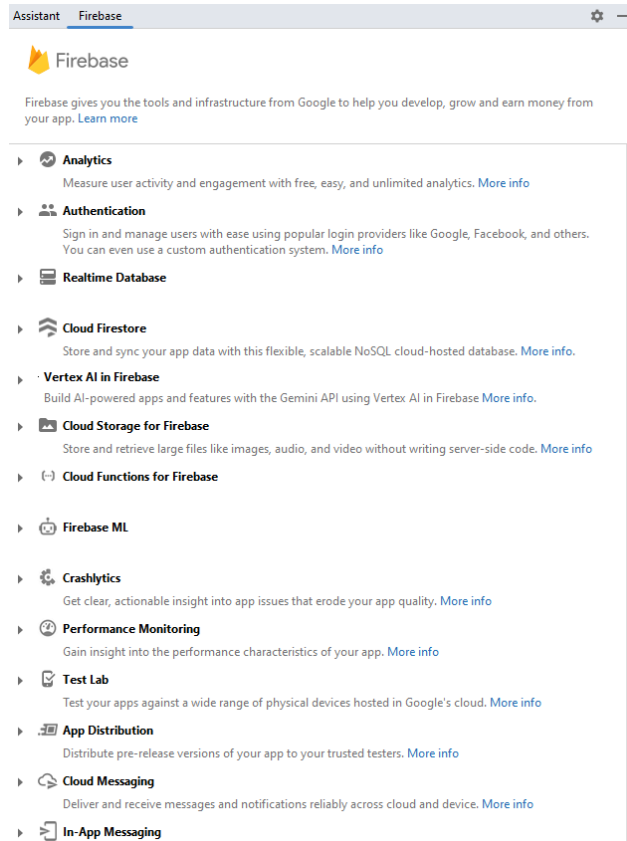
Firestore Realtime Database

- Las apps conectadas comparten datos.
- Alojamiento en la nube (servicio en la nube propiedad de Google).
- Los datos se almacenan en formato JSON.
- Para cada cliente conectado, los datos se sincronizan en tiempo real.



Firebase en Android Studio (1/2)

- Conectar la app con Firebase a través del asistente de Android Studio: **Tools > Firebase.**
- Añadir al fichero **build.gradle.kts** las dependencias necesarias.
- Hacer uso del asistente de Android Studio para Firebase en las distintas configuraciones del proyecto (autenticación, mensajería...).



Firestore en Android Studio (2/2)

- Enlaces útiles:
 - [Conexión a Firestore desde Android Studio.](#)
 - [Documentación básica de Firestore.](#)
 - [Conecta tu app a Firestore.](#)