cpifp Los Enlaces

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

2º DESARROLLO DE APLICACIONES MULTIPLATAFORMA

TEMA 7. PERSISTENCIA DE DATOS

T7. PERSISTENCIA DE DATOS

0. PERSISTENCIA DE DATOS

Los programas, ya sean de escritorio, web o móvil, necesitan guardar datos de una ejecución a otra para su correcto funcionamiento. Quizá necesiten almacenar opciones de usuario, quizá guarden logs de depuración con errores y alertas, quizá estemos hablando de un juego en el que debemos guardar los jugadores y los puntos que han conseguido en cada partida.

Si lo que necesitamos guardar no es más que un pequeño conjunto de pares clave-valor, utilizamos la librería DataStore. Un objeto de este tipo utilizará un archivo para escribir y leer pares clave-valor.

T7. PERSISTENCIA DE DATOS

1. APLICACIÓN SETTINGS

Vamos a preparar una actividad que simule una pantalla de configuración típica para cualquier aplicación.

Como siempre, incluimos un nuevo directorio en nuestro proyecto, al que llamaremos *Settings*. Y le añadimos una *Empty Views Activity* de nombre *SettingsActivity*.

Abrimos *MenuActivity.kt* y *activity_menu.xml* e incluimos el botón necesario para acceder a esta nueva actividad.

1. APLICACIÓN SETTINGS

Incluimos el método ViewBinding. Aquí tienes las líneas necesarias pero también puedes probar a insertarlo sin copiar el código y así puedes entrenar para memorizarlo.

```
class SettingsActivity : AppCompatActivity() {

    private lateinit var binding: ActivitySettingsBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding=ActivitySettingsBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

T7. PERSISTENCIA DE DATOS

1. APLICACIÓN SETTINGS

Ahora vamos a pasar al diseño. Es bueno seguir patrones de diseño que hagan la experiencia de usuario más cómoda.

Existen muchas páginas con consejos que pueden ayudarnos a configurar la interfaz de usuario de nuestras aplicaciones de forma que ésta resulte agradable e intuitiva. Un ejemplo de ello es:

https://m2.material.io/design/platform-guidance/android-settings.html

En su sección *Android Settings* podemos aprovechar recomendaciones sobre algunos elementos típicos para una pantalla de configuración.

2. LAYOUT PRINCIPAL

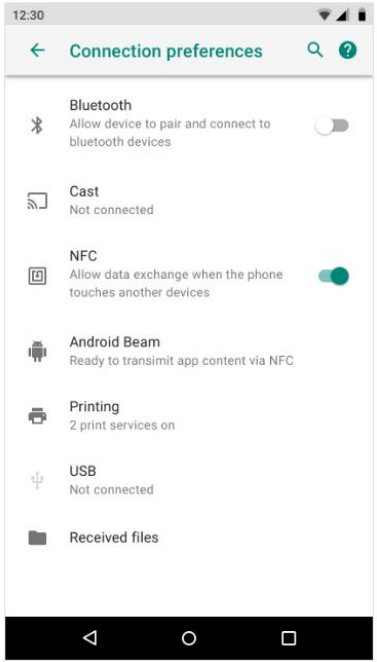
Vamos a crear una pantalla con diferentes opciones de configuración alineadas verticalmente para la que usaremos un *LinearLayout*.

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".Settings.SettingsActivity">

</LinearLayout>
```

2. LAYOUT PRINCIPAL

Nos vamos a fijar en la imagen de la derecha y vamos a crear elementos que contengan un icono, textos y un checkbox. Necesitamos otro LinearLayout horizontal.

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center_vertical"
    android:orientation="horizontal">

</LinearLayout>
```



2. LAYOUT PRINCIPAL

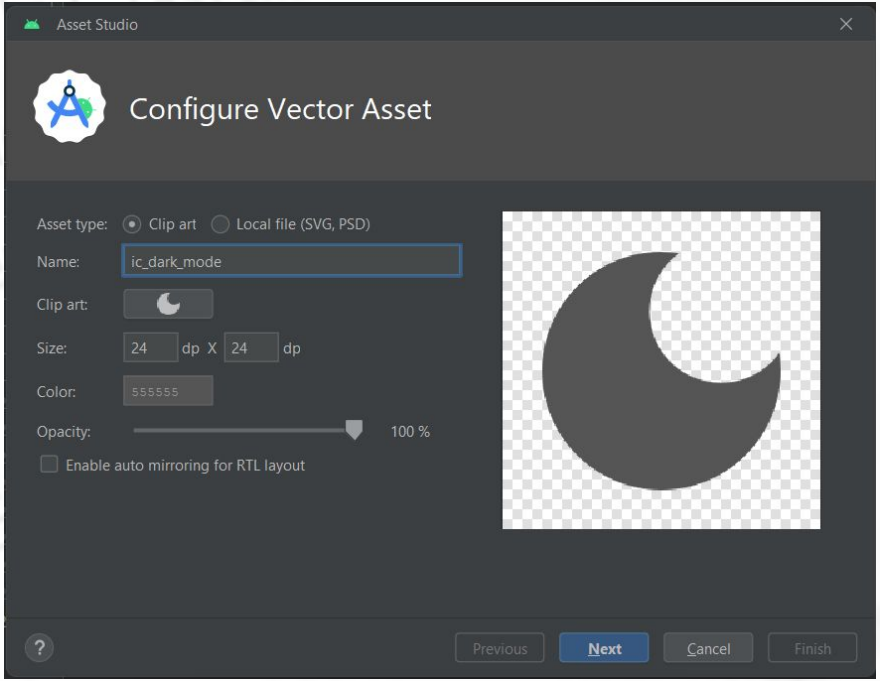
Ahora incluimos un *ImageView* para el icono que situaremos a la izquierda. De momento, podemos usar iconos predefinidos en AndroidStudio.

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:src="@drawable/ic_dark_mode" />
```

2. LAYOUT PRINCIPAL

Para usar iconos propios de AndroidStudio podemos incluirlos en el directorio */drawable* desde *New → Vector Asset*. En la ventana emergente seleccionamos *Clip art* y buscamos el icono para el modo oscuro.

Lo renombramos *ic_dark_mode* y configuramos color al hexadecimal 555555.



2. LAYOUT PRINCIPAL

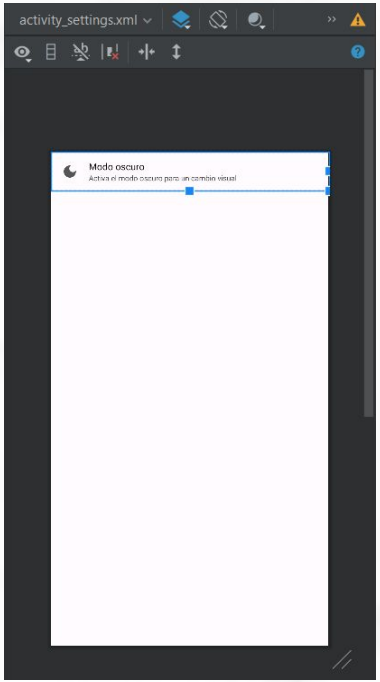
Lo siguiente es añadir dos TextViews alineados verticalmente, luego necesitamos un nuevo LinearLayout que los contenga.

```
<LinearLayout
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:orientation="vertical">

</LinearLayout>
```

2. LAYOUT PRINCIPAL

Estos TextViews seguirán el patrón de diseño que hemos visto en m2.material.io.

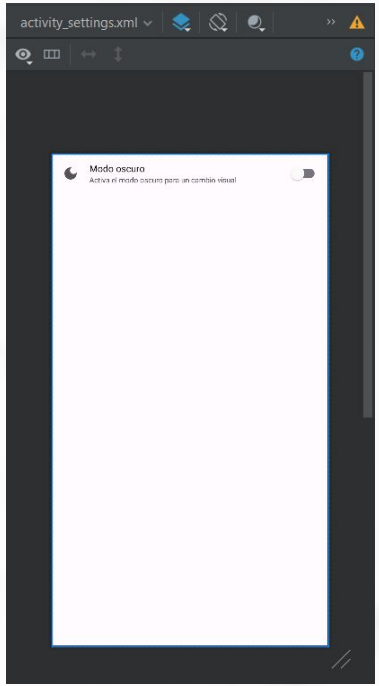
```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Modo oscuro"
    android:textColor="@color/black" />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Activa el modo oscuro para un cambio visual"
    android:textSize="11sp" />
```



2. LAYOUT PRINCIPAL

A la seguida del último LinearLayout insertamos ahora un nuevo elemento que simula un botón tipo *switch*. Puede que inicialmente no se muestre, no te preocupes.

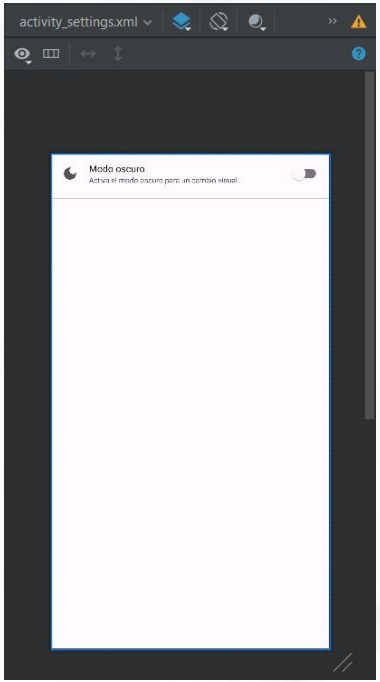
```
<com.google.android.material.switchmaterial.SwitchMaterial
    android:id="@+id/switchDarkMode"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginHorizontal="16dp"
    app:thumbTint="@color/white" />
```



2. LAYOUT PRINCIPAL

Y por último, vamos a insertar otro elemento nuevo, un separador (*divider*), que es otra de las recomendaciones de m2.material.io para resaltar distintas secciones.

```
<com.google.android.material.divider.MaterialDivider
    android:layout_width="match_parent"
    android:layout_height="1dp"
    android:layout_marginVertical="8dp" />
```

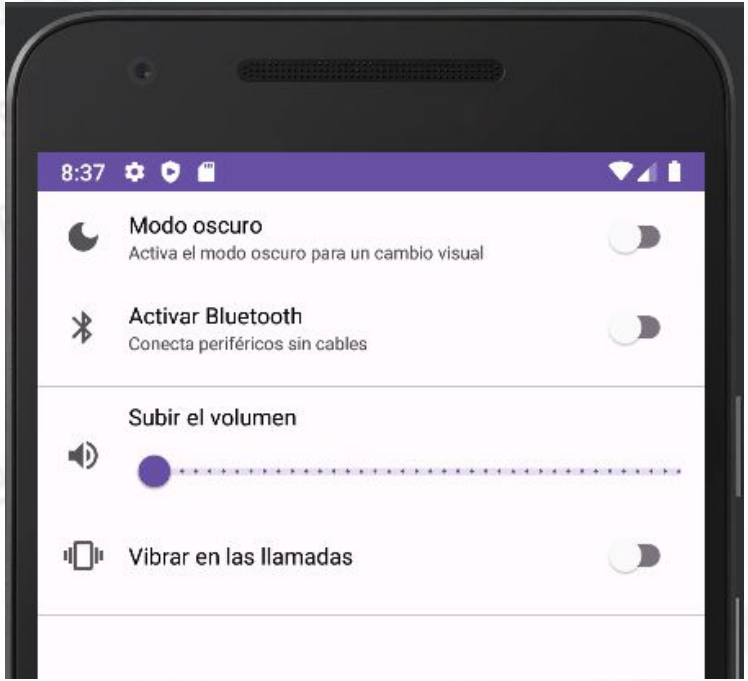


2. LAYOUT PRINCIPAL

Con todo esto, ya tienes las herramientas suficientes para terminar el diseño mostrado en la imagen de la derecha.

Para la sección *Subir el volumen* debes usar un RangeSlider como el que implementamos en la actividad IMCcalculator, con valores de 0 a 100.

Todos los iconos los puedes encontrar en *Clip Art* de *Vector Asset*.



3. LÓGICA

Lo primero que debemos hacer es incluir la librería DataStore en las dependencias de nuestro archivo *build.gradle.kts (Module: app)*. Tras incluirla, debes sincronizar pulsando en *Sync. now* arriba a la derecha.

```
//Picasso
implementation ("com.squareup.picasso:picasso:2.8")

//DataStore
implementation ("androidx.datastore:datastore-preferences:1.0.0")

    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
}
```

3. LÓGICA

Esto crea una especie de base de datos y, para acceder a ella, incluimos una función de extensión que nos permite crear métodos para cualquier tipo de elemento de nuestro layout.

Lo ideal sería construirla en un archivo aparte, pero para este ejemplo sencillo la añadimos en nuestra actividad principal, fuera de la clase *SettingsActivity* (ojo con el import de ***Preferences***, debe ser de **DataStore**).

```
val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = "settings")
//Utilizamos "by" en vez de "=" para generar una única instancia de la clase (singleton)

class SettingsActivity : AppCompatActivity() {
    private lateinit var binding: ActivitySettingsBinding
```

3. LÓGICA

Vamos a crear entonces una función que almacene el volumen seleccionado por el usuario mediante el RangeSlider, que será un valor tipo `Int` entre 0 y 100.

```
    binding = ActivitySettingsBinding.inflate(layoutInflater)
    setContentView(binding.root)
}

//El comando dataStore.edit debe ir dentro de una corrutina (suspend)
private suspend fun saveVolume(value: Int) {
    dataStore.edit {
        it[intPreferencesKey("volume_lvl")] = value
    }
}
```

3. LÓGICA

Con esta función estamos editando la “base de datos” y añadiendo o modificando el elemento `it["volume_lvl"]`. Podemos nombrar el array como queramos con la siguiente modificación:

```
    binding = ActivitySettingsBinding.inflate(layoutInflater)
    setContentView(binding.root)
}

//El comando dataStore.edit debe ir dentro de una corrutina (suspend)
private suspend fun saveVolume(value: Int) {
    dataStore.edit { preferences ->
        preferences[intPreferencesKey(VOLUME_LVL)] = value
    }
}
```

3. LÓGICA

Además, vamos a crearnos ya un *companion object* con todas las claves que vamos a utilizar para el array *preferences*. Como siempre, al inicio del código y dentro de nuestra clase principal.

```
class SettingsActivity : AppCompatActivity() {

    companion object {
        const val VOLUME_LVL = "volume_lvl"
        const val KEY_BLUETOOTH = "key_bluetooth"
        const val KEY_VIBRATION = "key_vibration"
        const val KEY_DARK_MODE = "key_dark_mode"
    }

private lateinit var binding: ActivitySettingsBinding
```

3. LÓGICA

Ahora vamos a crear nuestra función `initUI()` que invocaremos desde el método *onCreate* para interactuar con el layout. Para el RangeSlider implementamos el método *setOnChangeListener* de IMCapp.

```
private fun initUI() {
    binding.rsVolume.addOnChangeListener {  , value,  _ ->
        Log.i("Volumen", "El valor es $value")
        CoroutineScope(Dispatchers.IO).launch {
            saveVolume(value.toInt())
        }
    }
}
```

3. LÓGICA

Para los botones tipo *switch* del resto de opciones de configuración para el usuario, vamos a crear una única función que almacene un valor booleano y a la que le pasemos la clave del array de DataStore.

```
private suspend fun saveOptions(key: String, value: Boolean) {
    dataStore.edit { preferences ->
        preferences[booleanPreferencesKey(key)] = value
    }
}
```

3. LÓGICA

Así, dentro de la función `initUI()`, solo tendremos que invocar a dicha función dentro de una corrutina pasándole la clave correspondiente e implementando el método *setOnCheckedChangeListener* del *switchbutton*.

```
binding.switchBluetooth.setOnCheckedChangeListener { _, value ->
    CoroutineScope(Dispatchers.IO).launch {
        saveOptions(KEY_BLUETOOTH, value)
    }
}
binding.switchVibration.setOnCheckedChangeListener { _, value ->
    CoroutineScope(Dispatchers.IO).launch {
        saveOptions(KEY_VIBRATION, value)
    }
}
```

3. LÓGICA

Para el *switchbutton* (SwitchMaterial) que activa o desactiva el modo oscuro de la aplicación habrá que definir dos nuevas funciones que implementen dichas acciones.

```
binding.switchDarkMode.setOnCheckedChangeListener { _, value ->
    if(value){
        enableDarkMode()
    }else{
        disableDarkMode()
    }
    CoroutineScope(Dispatchers.IO).launch {
        saveOptions(KEY_DARK_MODE, value)
    }
}
```

3. LÓGICA

Aplicar el modo oscuro se hace a través del objeto *delegate* de la librería AppCompat. Ojo al importar los valores *MODE_NIGHT_YES* y *MODE_NIGHT_NO*, hay que hacerlo desde la clase AppCompatDelegate.

```
private fun enableDarkMode(){
    AppCompatDelegate.setDefaultNightMode(MODE_NIGHT_YES)
    delegate.applyDayNight()
}

private fun disableDarkMode(){
    AppCompatDelegate.setDefaultNightMode(MODE_NIGHT_NO)
    delegate.applyDayNight()
}
```

3. LÓGICA

Ya hemos guardado los valores seleccionados por el usuario pero ahora debemos recuperarlos. Para ello, tomamos los datos almacenados en DataStore de su atributo *data.map*, con valores por defecto en caso NULL.

```
private fun getSettings() {
    return dataStore.data.map { preferences ->
            preferences[intPreferencesKey(VOLUME_LVL)] ?:50
            preferences[booleanPreferencesKey(KEY_BLUETOOTH)] ?:true
            preferences[booleanPreferencesKey(KEY_DARK_MODE)] ?:false
            preferences[booleanPreferencesKey(KEY_VIBRATION)] ?:true
    }
}
```

3. LÓGICA

Pero debemos indicarle a la función qué tipo de respuesta va a devolver. Para ello hacemos uso de *Flow*, que es una especie de listener para datos almacenados. El problema es que solo es capaz de devolver un valor.

```
private fun getSettings(): Flow<Boolean> {
    return dataStore.data.map { preferences ->
            preferences[intPreferencesKey(VOLUME_LVL)] ?: 50
            preferences[booleanPreferencesKey(KEY_BLUETOOTH)] ?: true
            preferences[booleanPreferencesKey(KEY_DARK_MODE)] ?: false
            preferences[booleanPreferencesKey(KEY_VIBRATION)] ?: true
    }
}
```

3. LÓGICA

Entonces tenemos que crear un objeto que contenga todos los valores de DataStore que queramos tomar para pasarlos a través de *Flow*. Lo incluimos en la función y lo definimos en una nueva *Kotlin Class*.

```
private fun getSettings(): Flow<SettingsModel?> {
    return dataStore.data.map { preferences ->
        SettingsModel(
            volume = preferences[intPreferencesKey(VOLUME_LVL)] ?: 50,
            bluetooth = preferences[booleanPreferencesKey(KEY_BLUETOOTH)] ?: true,
            darkMode = preferences[booleanPreferencesKey(KEY_DARK_MODE)] ?: false,
            vibration = preferences[booleanPreferencesKey(KEY_VIBRATION)] ?: true
        )
    }
}
```

3. LÓGICA

Esta clase, llamada *SettingsModel.kt*, será una *data class* con los cuatro valores posibles para la configuración de usuario. Es decir, un valor tipo `int` y tres valores tipo `boolean`.

```
data class SettingsModel(
    var volume: Int,
    var bluetooth: Boolean,
    var darkMode: Boolean,
    var vibration: Boolean
)
```

3. LÓGICA

Para consumir este *Flow*, vamos al método *onCreate* y, antes de la función `initUI()`, implementamos el método *collect* de DataStore. Pero éste se debe ejecutar en un hilo secundario, igual que con *edit*.

```
CoroutineScope(Dispatchers.IO).launch {
    getSettings().collect { settingsModel ->
        binding.switchVibration.isChecked = settingsModel.vibration
        binding.switchBluetooth.isChecked = settingsModel.bluetooth
        binding.switchDarkMode.isChecked = settingsModel.darkMode
        binding.rsVolume.setValues(settingsModel.volume.toFloat())
    }
}
```

3. LÓGICA

Sin embargo, es en el hilo principal donde debemos configurar los valores de las distintas Views. Y no está demás incluir un filtro para asegurarnos que no recibe nunca un valor NULL del objeto *SettingsModel*.

```
CoroutineScope(Dispatchers.IO).launch {
    getSettings().collect { settingsModel ->
        if (settingsModel != null) {
            runOnUiThread {
                binding.switchVibration.isChecked = settingsModel.vibration
                binding.switchBluetooth.isChecked = settingsModel.bluetooth
                binding.switchDarkMode.isChecked = settingsModel.darkMode
                binding.rsVolume.setValues(settingsModel.volume.toFloat())
            }
        }
```

3. LÓGICA

Si intentamos ejecutar la aplicación ahora, cuando modifiquemos uno de los valores de configuración, el *Flow* enviará los datos modificados continuamente. Para solucionarlo, vamos a poner un filtro.

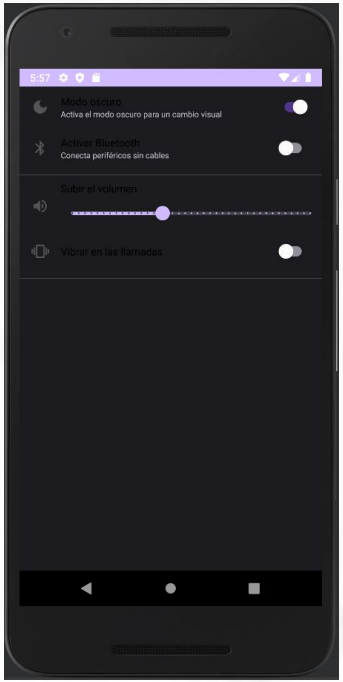
```
private lateinit var binding: ActivitySettingsBinding
private var firstTime:Boolean = true
...
    CoroutineScope(Dispatchers.IO).launch {
        getSettings().filter { firstTime }.collect { settingsModel ->
            if (settingsModel != null) {
                runOnUiThread {
                    ...
                    binding.rsVolume.setValues(settingsModel.volume.toFloat())
                    firstTime = !firstTime
```

4. MODO OSCURO

Con esto ya tenemos nuestra pantalla de opciones plenamente funcional.

El problema es que los TextViews que hemos definido para los títulos de las distintas configuraciones tienen un color de fuente @color/black.

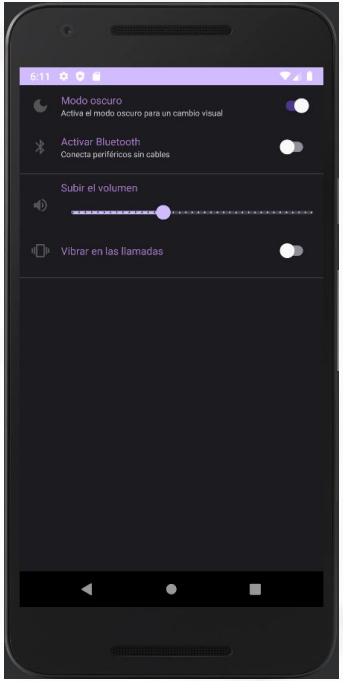
Si quitamos el atributo *textColor*, Android tomará un color por defecto que cambiará con el tema de la aplicación. Pero si queremos poner un color personalizado, podemos definir una paleta de colores para el modo oscuro.



4. MODO OSCURO

Si vamos a la vista de proyecto (*Project*), en la ruta *aplicacionesPMDM/app/src/main/res* podemos ver los directorios */values* y */values-night*.
Dentro de este último podemos crear un nuevo *Values Resource File* que se llame *colors.xml* y configurar un color personalizado para los títulos.

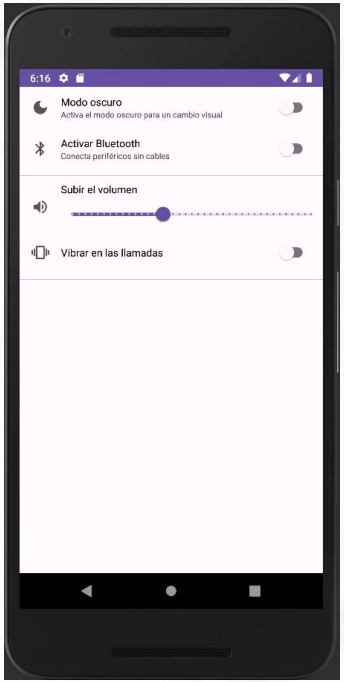
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!--Settings-->
    <color name="settings_text_title">#A177C5</color>
</resources>
```



4. MODO OSCURO

Haremos lo mismo en el archivo *colors.xml* dentro del directorio */values*, pero cambiando el color a negro y conservando el mismo nombre (*name*).
Ahora, a cada TextView del layout *activity_settings.xml* deberemos cambiarle el color de texto al que acabamos de definir: @colors/settings_text_title.

```
    <!--Settings-->
    <color name="settings_text_title">#FF000000</color>

</resources>
```



4. MODO OSCURO

También podemos definir una paleta de colores para un *theme*, como vimos en la aplicación *BoardgamesApp*.

En *HelloApp* y *MessageApp* podemos ver cómo se aplica el modo oscuro, pero en la aplicación de juegos de mesa no ocurre lo mismo.

Define un tema oscuro para esta aplicación con los colores que tú elijas. Ten en cuenta que se ha de buscar una paleta de colores con poca luminosidad. O considera el tema oscuro como el ya definido y crea un modo claro.