IES DR LLUÍS SIMARRO
ENSENYAMENT PÚBLIC
XATIVA

Programación multimedia y dispositivos móviles

UD07. Diseño de menús y elementos multimedia.

Desarrollo de Aplicaciones Multiplataforma

Anna Sanchis Perales
Enric Giménez Ribes

ÍNDICE

1. OBJETIVOS

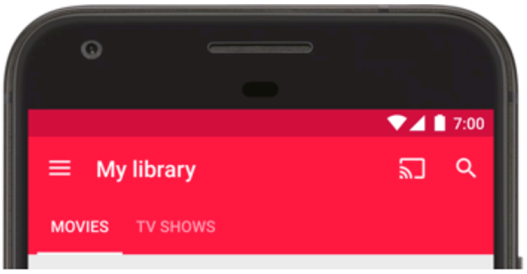
Los objetivos a conocer y comprender son:

- Controlar el diseño de la barra de acciones y menús en Android.
- Conocer cómo se trabaja con la Toolbar.
- Conocer cómo se trabaja con el BottomNavigationView.
- Estudiar nuevos elementos de diseño: pestañas y menú lateral.
- Crear pantallas de bienvenida (pantalla Splash).
- Ver la viabilidad de trabajar con diferentes menús.

2. APPBAR

La **AppBar** (anteriormente se llamaba **ActionBar**) fue introducida a partir de la versión 3.0 de Android. Es la **barra que aparece en la parte superior de cada Activity** y que suele contener:

- El nombre de la aplicación.
- El icono de nuestra aplicación (ya no se recomienda en las últimas versiones ponerlo).
- Acciones de menú (lo veremos más adelante).
- Incluso se usa para la navegación, ya sea por pestañas o lista desplegable.
- También puede aparecer a la izquierda del título iconos indicativos de la existencia de menú lateral deslizante (navigation drawer) o el botón de navegación hacia atrás/arriba, pero esto ya lo veremos más adelante.



Tenemos más información:

- https://developer.android.com/training/appbar/

A partir de aquí, podemos hacer uso de la AppBar (o ActionBar) de dos formas diferentes.

1. La primera de ellas, más sencilla aunque menos flexible, consiste en utilizar la **AppBar** por defecto que se añade a nuestras actividades por tan sólo utilizar un tema (theme) determinado en la aplicación y extender nuestras actividades de una clase específica.
2. La segunda, más flexible y personalizable aunque requiere más código, consiste en utilizar el nuevo componente **Toolbar** disponible desde la llegada de Android 5.0 (aunque compatible con versiones anteriores).

En este punto nos centraremos en la segunda de las opciones, por ser la más flexible y la que recomienda el developer de Android.

3. TOOLBAR

La forma más flexible y personalizable de añadir una AppBar a una aplicación es utilizar el nuevo componente **Toolbar** proporcionado por la librería appcompat. **De esta forma podemos incluir de forma explícita la AppBar en nuestros layouts XML como si fuera cualquier otro control, y no sólo en la parte superior de la pantalla, sino también en cualquier otro lugar de la aplicación donde queramos utilizar esta funcionalidad de barra de acciones.**

Ejemplo Toolbar

Veremos primero cómo utilizar el componente Toolbar a modo de AppBar.

Vamos a crear un nuevo proyecto con el que trabajemos con el Toolbar:

1. Creamos un nuevo proyecto denominado **Tema7App1**.
2. Lo primero que debemos asegurar es que nuestro proyecto incluye la última versión de la librería de soporte appcompat, en la sección de dependencias del fichero build.gradle:

```
dependencies {
   implementation 'androidx.appcompat:appcompat:1.6.1'
   implementation 'com.google.android.material:material:1.7.0'
   ...
}
```

3. A continuación, nos aseguramos de que tenemos desactivada la actionbar en el thema que usa nuestra app. Para ello, vamos al fichero /res/values/themes.xml y veremos que nuestro tema extiende de alguno de los siguientes (dependiendo si queremos partir del tema oscuro o claro):
    - Theme.MaterialComponents.NoActionBar
    - Theme.MaterialComponents.Light.NoActionBar

4. Hecho esto, ya podemos modificar el layout de nuestra actividad para incluir la AppBar utilizando el nuevo componente Toolbar. Veamos cómo quedaría el código y después comentamos los detalles más importantes:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   tools:context=".MainActivity">

   <androidx.appcompat.widget.Toolbar
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:id="@+id/appbar"
       android:layout_height="wrap_content"
       android:layout_width="match_parent"
       android:minHeight="?attr/actionBarSize"
       android:background="?attr/colorPrimary"
       android:elevation="4dp"
       android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
       app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

   <!-- Resto de la interfaz de usuario -->

   <TextView android:text="Hello World!"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content" />
</LinearLayout>
```

Utilizaremos un LinearLayout como contenedor principal, sin márgenes interiores (para que el Toolbar ocupe toda la pantalla en la parte superior), y en su interior incluye como primer control el Toolbar que actuará como AppBar de la actividad. Entre las propiedades asignadas, además de las habituales id, layout_height y layout_width, vemos las siguientes:

- **android:minHeight.** Asignando esta propiedad al alto estándar de un AppBar (?attr/actionBarSize) conseguimos que los botones de acción y menú de overflow queden siempre en la parte superior de la barra de herramientas aunque incrementemos su tamaño mediante layout_height (referencia).
- **android:background.** Asignaremos a esta propiedad el valor ?attr/colorPrimary de forma que se utilice como color de la barra de herramientas el que hemos definido como colorPrimary en el tema de la aplicación (/res/values/themes.xml).
- **android:elevation.** Esta propiedad define la elevación del componente, lo que determina la sombra que proyectará sobre el elemento inferior. La elevación estándar de la AppBar definida en las guías de diseño es de 4dp. Esta propiedad tan

sólo tiene efecto cuando la aplicación se ejecuta sobre Android 5.0 o superior, aunque más adelante veremos cómo solucionarlo.

- **android:theme.** Mediante esta propiedad definimos el tema a utilizar por la Toolbar (y que heredarán sus controles hijos). No debemos confundir esto con el tema definido a nivel de aplicación en el fichero themes.xml. En la Toolbar utilizaremos el tema oscuro asignando la propiedad app:theme, en este caso con el tema ThemeOverlay.AppCompat.Dark.ActionBar.
- **app:popupTheme.** Esta propiedad la utilizaremos sólo si es necesario. En nuestro caso particular lo es, para "arreglar" un efecto colateral de utilizar el tema oscuro para la Toolbar. Y es que utilizar este tema también tiene como efecto que el menú de overflow aparezca de color oscuro. Para conseguir que la barra sea oscura pero el menú claro podemos asignar un tema específico sólo a este menú utilizando la propiedad app:popupTheme, en nuestro al tema ThemeOverlay.AppCompat.Light.

5. Con esto ya tendríamos finalizado el layout XML de la actividad principal con su AppBar correspondiente, por supuesto a falta de incluir el resto de controles necesarios para nuestra aplicación, que en este caso dejaremos el TextView por defecto.
6. Por su parte, el menú de overflow se definiría sobreescribiendo el método onCreateOptionsMenu(menu: Menu?) de la actividad que queremos que lo muestre. En este evento deberemos inflar el menú, primero obtendremos una referencia al inflater mediante el menuInflater y posteriormente generamos la estructura del menú llamando a su método inflate() pasándole como parámetro el ID del menú definido en XML (se creará en el siguiente punto), que en nuestro caso será R.menu.menu_main. Por último, devolveremos el valor true para confirmar que debe mostrarse el menú. Así que, en el ficheros MainActivity.kt, haremos lo siguiente:

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
   menuInflater.inflate(R.menu.menu_main, menu)
   return true
}
```

7. Ahora vamos a crear alguna opción dentro del Toolbar. Para ello hemos de crear un menú que será el que inflemos, y así podremos mostrarlo. Por ello comenzamos creando (si no existe) la carpeta /res/menu.

8. Tras crear la carpeta que va a crear el menú, crearemos un fichero de recursos XML pulsando botón derecho sobre la carpeta y seleccionando New > Menu Resource File. Le asignamos de nombre menu_main.xml, de forma que será el menú del Toolbar de la pantalla principal.

```
<menu 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"
     tools:context=".MainActivity">

      <item android:id="@+id/action_settings"
         android:title="@string/action_settings"
         android:orderInCategory="100"
          app:showAsAction="never" />

      <item android:id="@+id/action_buscar"
          android:title="@string/action_buscar"
          android:icon="@android:drawable/ic_menu_search"
          android:orderInCategory="100"
          app:showAsAction="ifRoom" />

      <item android:id="@+id/action_nuevo"
          android:title="@string/action_nuevo"
          android:icon="@android:drawable/ic_menu_add"
          android:orderInCategory="100"
          app:showAsAction="ifRoom|withText" />
</menu>
```

El menú se define mediante un elemento raíz <menu> y contendrá una serie de elementos <item> que representarán cada una de las opciones. Los elementos <item> por su parte podrán incluir varios atributos que lo definan, entre los que destacan los siguientes:

- **android:id.** El ID identificativo del elemento, con el que podremos hacer referencia dicha opción.
- **android:title.** El texto que se visualizará para la opción.
- **android:icon.** El icono asociado a la acción.
- **android:orderInCategory.** A mayor valor, menos prioridad. Si los ítems aparecen en la AppBar, aparecerán de izquierda a derecha, primero se mostrará el que tenga un menor valor en esta propiedad y así los siguientes. En cambio, si los ítems aparecen en el menú overflow será lo mismo pero de arriba abajo en función del valor.
- **android:showAsAction.** Si se está mostrando una AppBar, este atributo indica si la

opción de menú se mostrará como botón de acción o como parte del menú de overflow. Puede tomar varios valores:

- ifRoom. Se mostrará como botón de acción sólo si hay espacio disponible.
- withText. Se mostrará el texto de la opción junto al icono en el caso de que éste se esté mostrando como botón de acción.
- never. La opción siempre se mostrará como parte del menú de overflow.
- always. La opción siempre se mostrará como botón de acción. Este valor puede provocar que los elementos se solapen si no hay espacio suficiente para ellos.

9. Modificamos el fichero /res/values/strings.xml para añadir los nuevos valores:

```
<resources>
    <string name="app_name">Tema6App2</string>
    <string name="action_settings">Settings</string>
    <string name="action_buscar">Search</string>
    <string name="action_nuevo">Add</string>
</resources>
```

10. Ahora nos queda definir y personalizar las pulsaciones sobre los elementos.

Para esto sobrescribiremos el método OnOptionsItemSelected() de la actividad, donde consultaremos la opción de menú que se ha pulsado mediante el método getItemId() de la opción de menú recibida como parámetro y actuaremos en consecuencia. En nuestro caso tan sólo mostraremos un Toast con un mensaje, por lo que nos vamos a la clase MainActivity.kt y añadimos lo siguiente:

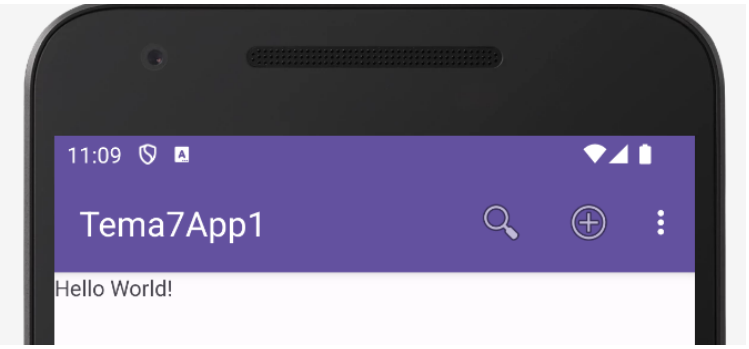
```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
   return when(item.itemId){
       R.id.action_settings ->{
           showToast("Opción settings seleccionada")
           true
       }
       R.id.action_buscar ->{
           showToast("Opción search seleccionada")
           true
       }
       R.id.action_nuevo ->{
           showToast("Opción add seleccionada")
           true
       }
```

```
        else -> super.onOptionsItemSelected(item)
    }
}

private fun showToast(message: String) {
    Toast.makeText(this, message, Toast.LENGTH_SHORT).show()

}
```

11. Pero nos queda aún un paso importante. En nuestro código debemos indicar que esta Toolbar actuará como AppBar de la actividad. Para ello, en el método onCreate() de la actividad MainActivity.kt haremos una llamada a setSupportActionBar() con la referencia a la Toolbar:

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

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

        val toolBar: androidx.appcompat.widget.Toolbar =
findViewById<androidx.appcompat.widget.Toolbar>(R.id.appbar)
        setSupportActionBar(toolBar)
    }

...
```

12. Si ejecutamos la aplicación obtenemos lo siguiente:



Include - Para no reescribir el ToolBar

13. Para no tener que reescribir la definición completa del toolbar en todas nuestras actividades (tendríamos que estar repitiendo la mayor parte de este código en todas las actividades) también podemos hacer uso de la cláusula include. Para ello, declaramos primero el toolbar en un layout XML independiente, por ejemplo en un fichero llamado /res/layout/toolbar.xml, por lo que crearemos dicho fichero con el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.appcompat.widget.Toolbar
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   android:layout_height="wrap_content"
   android:layout_width="match_parent"
   android:minHeight="?attr/actionBarSize"
   android:background="?attr/colorPrimary"
   android:elevation="4dp"
   android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
   app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />
```

14. Y posteriormente incluiremos este layout toolbar.xml en el layout activity_main.xml de nuestras actividades haciendo referencia a él mediante include:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   tools:context=".MainActivity">

   <include android:id="@+id/appbar"
       layout="@layout/toolbar" />

   <!-- Resto de la interfaz de usuario -->

   <TextView android:text="Hello World!"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content" />
</LinearLayout>
```

15. Como se puede observar, en este caso definimos la propiedad android:id del control en en el include, y no en el layout independiente del toolbar.

16. Ahora es cuestión de ejecutar la aplicación y ver que el resultado es el mismo que en el ejemplo anterior.

Personalizar la ToolBar

Podemos personalizar la ToolBar muy fácilmente, por ejemplo modificando su tamaño o añadiendo controles en su interior. Para crear por ejemplo una AppBar extendida que además muestra dos líneas de texto con título y subtítulo podríamos modificar de la siguiente forma nuestro fichero toolbar.xml en nuestro proyecto.

1. Abrimos el proyecto **Tema7App1**.
2. Modificamos el fichero /res/layout/toolbar.xml de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.appcompat.widget.Toolbar
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:layout_height="128dp"
       android:layout_width="match_parent"
       android:minHeight="?attr/actionBarSize"
       android:background="?attr/colorPrimary"
       android:elevation="4dp"
       android:gravity="bottom"
       android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
       app:popupTheme="@style/ThemeOverlay.AppCompat.Light">

       <LinearLayout
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:orientation="vertical"
          android:paddingBottom="16dp" >

          <TextView android:id="@+id/txtAbTitulo"
              android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:text="@string/Título"
              style="@style/TextAppearance.AppCompat.Widget.ActionBar.Title" />

          <TextView android:id="@+id/txtAbSubTitulo"
              android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:text="@string/Subtítulo"
              style="@style/TextAppearance.AppCompat.Widget.ActionBar.Subtitle" />

       </LinearLayout>

</androidx.appcompat.widget.Toolbar>
```

Se han asignado dos nuevas propiedades en el control Toolbar. En primer lugar hemos incrementado el alto del control a 128dp, y además hemos establecido la propiedad gravity al valor “bottom”, de forma que los componentes que añadamos en su interior se alinean sobre el borde inferior.

Adicionalmente, dentro del elemento Toolbar hemos creado un LinearLayout que va a contener dos etiquetas de texto para mostrar el título y el subtítulo, como si se tratara de cualquier otro contenedor. Lo único a destacar es que he utilizado dos estilos predefinidos para estas etiquetas (AppCompat.Widget.ActionBar.Title y AppCompat.Widget.ActionBar.Subtitle) aunque podría utilizarse cualquier formato para estos elementos.

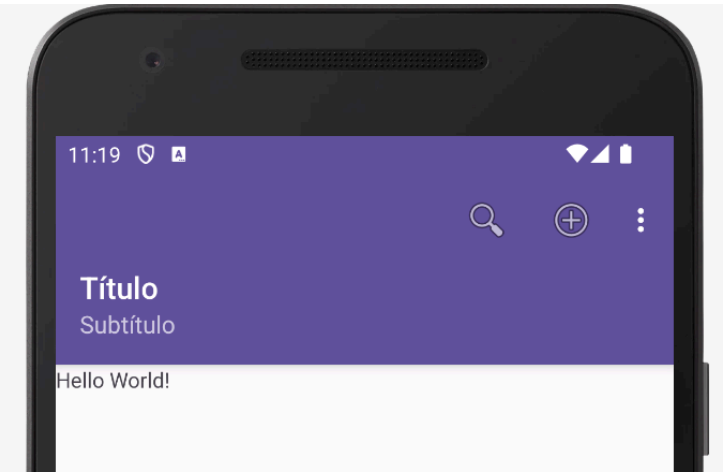
3. Nos vamos al fichero /res/values/strings.xml y definimos los siguientes valores:

```
<resources>
   <string name="app_name">Tem6App2</string>
   <string name="action_settings">Settings</string>
   <string name="action_buscar">Search</string>
   <string name="action_nuevo">Add</string>
   <string name="Título">Título largo AppBar</string>
   <string name="Subtítulo">Subtítulo AppBar</string>
</resources>
```

4. En MainActivity.kt deshabilitamos que se muestre el título de la app:

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

   val toolBar: androidx.appcompat.widget.Toolbar =
findViewById<androidx.appcompat.widget.Toolbar>(R.id.appbar)
   setSupportActionBar(toolBar)
   supportActionBar?.setDisplayShowTitleEnabled(false) // Esto oculta el título de la
app en la Toolbar
}
```

5. Si ejecutamos el proyecto obtenemos el siguiente resultado:



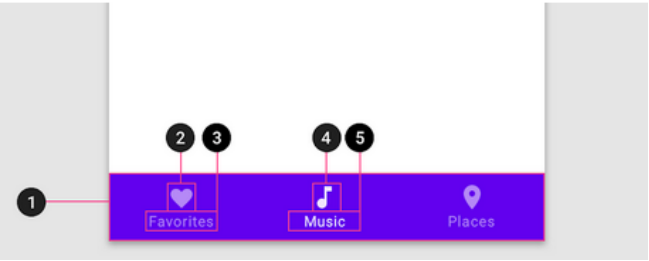
4. BOTTOM NAVIGATION

El componente **BottomNavigationView** representa una barra de navegación inferior estándar para nuestras aplicaciones. Es una implementación de Material Design y, por lo tanto, necesitamos tener la dependencia en nuestro archivo Gradle.

```
dependencies {
   implementation 'com.google.android.material:material:1.7.0'
   ...
}
```

Las barras de navegación inferiores facilitan a los usuarios explorar y cambiar entre las vistas de nivel superior con un solo toque. Deben utilizarse cuando una aplicación tiene de tres a cinco destinos de nivel superior.

Su aspecto es el siguiente:



1. Contenedor
2. Icono inactivo
3. Etiqueta de texto inactiva
4. Icono activo
5. Etiqueta de texto activa

El contenido de la barra se puede completar especificando un archivo de recursos de menú. El título, icono y estado habilitado de cada elemento del menú se utilizará para mostrar los elementos de la barra de navegación inferior. Los elementos del menú también se pueden utilizar para seleccionar mediante programación qué destino está activo actualmente.

- https://material.io/components/bottom-navigation

Vamos a crear un nuevo proyecto **Tema7App2:**

1. Primero, comencemos con el diseño XML (activity_main.xml) que contiene el BottomNavigationView:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <FrameLayout
       android:id="@+id/container"
       android:layout_width="0dp"
       android:layout_height="0dp"
       app:layout_constraintTop_toTopOf="parent"
       app:layout_constraintBottom_toTopOf="@id/bottomNavigation"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintEnd_toEndOf="parent" />

   <com.google.android.material.bottomnavigation.BottomNavigationView
       android:id="@+id/bottomNavigation"
       android:layout_width="0dp"
       android:layout_height="wrap_content"
       app:menu="@menu/bottom_navigation_menu"
       app:layout_constraintBottom_toBottomOf="parent"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintEnd_toEndOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

En este XML, hay un FrameLayout que actuará como contenedor del contenido dinámico y un BottomNavigationView que se muestra en la parte inferior. El atributo app:menu="@menu/bottom_navigation_menu" hace referencia a un archivo de menú que contiene las opciones del BottomNavigationView

2. Luego, en el archivo de menú (bottom_navigation_menu.xml), definiremos las opciones que se mostrarán en el BottomNavigationView:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item
       android:id="@+id/navigation_home"
       android:title="Home"
       android:icon="@drawable/ic_home" />

   <item
       android:id="@+id/navigation_dashboard"
       android:title="Dashboard"
       android:icon="@drawable/ic_dashboard" />

   <item
       android:id="@+id/navigation_notifications"
       android:title="Notifications"
       android:icon="@drawable/ic_notifications" />

</menu>
```

Crearemos para ello los pertinentes iconos dentro de la carpeta drawable.

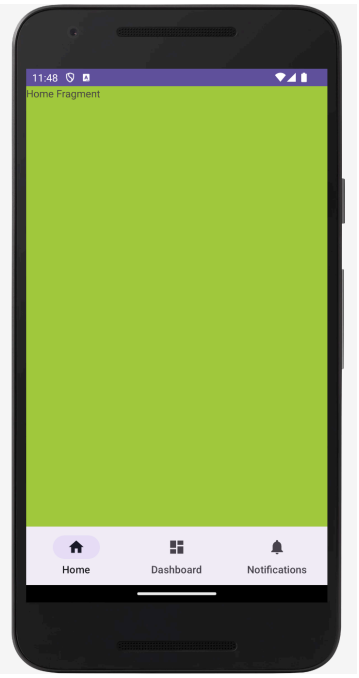
3. Hemos creado tres Blank Fragments, llamados DashboardFragment, HomeFragment y NotificationsFragment. El código Kotlin lo dejamos tal cual, solamente modificaremos el diseño xml, a continuación se muestra uno de ellos.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="@android:color/holo_blue_light"
   tools:context=".fragments.DashboardFragment">

   <!-- TODO: Update blank fragment layout -->
   <TextView
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:text="Dashboard Fragment" />

</FrameLayout>
```

4. Finalmente, en tu actividad en Kotlin (MainActivity.kt), puedes manejar las selecciones del BottomNavigationView:

```
class MainActivity : AppCompatActivity() {

   private lateinit var binding: ActivityMainBinding


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

       //Inicializamos con el HomeFragment
       supportFragmentManager.beginTransaction()
           .replace(R.id.container, HomeFragment())
           .commit()

       binding.bottomNavigation.setOnNavigationItemSelectedListener {
           it.isChecked = true
           when (it.itemId) {
               R.id.navigation_home -> {
                   // Acción al seleccionar Home
                   replaceFragment(HomeFragment())
               }
               R.id.navigation_dashboard -> {
                   // Acción al seleccionar Dashboard
                   replaceFragment(DashboardFragment())
               }
               R.id.navigation_notifications -> {
                   // Acción al seleccionar Notifications
                   replaceFragment(NotificationsFragment())
               }
           }
           false
       }



   }

   private fun replaceFragment(fragment: Fragment) {
       supportFragmentManager.beginTransaction()
           .replace(R.id.container, fragment)
           .commit()
   }

}
```

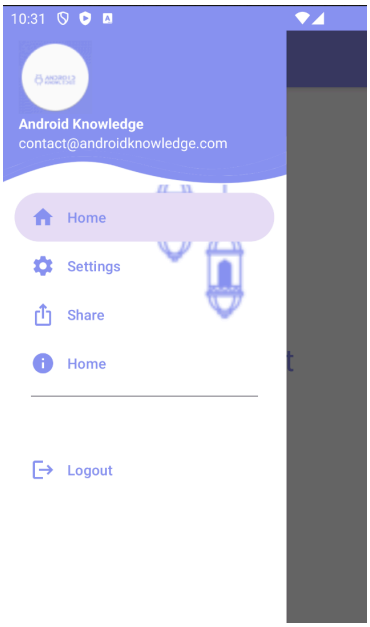
5. Si procedemos a la ejecución del proyecto obtenemos lo siguiente:



5. ELEMENTOS DE DISEÑO

5.2. Menú lateral (Navigation Drawer)

El **menú lateral (Navigation Drawer)** es una buena opción para aprovechar al máximo el tamaño de la pantalla y así se pueda visualizar mejor la información en la pantalla principal.



Para más información:

- https://material.io/components/navigation-drawer

Navigation Drawer

Tenemos un ejemplo de cómo incluir este tipo de menú en nuestras aplicaciones en el siguiente vídeo de Android Knowledge:

- https://www.youtube.com/watch?v=sSL6a_ivRVk&t=384s

Y aquí un enlace donde podemos ver el código del ejemplo:

https://androidknowledge.com/navigation-drawer-android-studio-kotlin/

5.3. Pantalla Splash

Las **pantallas Splash** son las pantallas de bienvenida a una App, aportan una mayor sensación de profesionalidad, tradicionalmente se muestran unos segundos a pantalla completa, y seguidamente, se abre el layout principal de la App.

Opción 1

Creamos una actividad nueva y esta actividad será la que se lance al abrir la aplicación.

```
class SplashActivity : AppCompatActivity() {
   private lateinit var binding: ActivitySplashBinding

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

       // Reproduce la animación Lottie
       binding.animationView.playAnimation()

       // Configura un temporizador para mostrar la pantalla de inicio durante un
cierto tiempo
       val splashScreenDuration = 3000 // 3 segundos
       val intent = Intent(this, WelcomeActivity::class.java)

       // Inicia la actividad principal después del tiempo especificado
       binding.animationView.postDelayed({
           startActivity(intent)
           finish()
       }, splashScreenDuration.toLong())
   }
}
```

La actividad tendrá un tiempo de espera, y luego por medio de una función anónima crea y lanza un Intent para cambiar de actividad, el finish() es necesario para quitar de la pila de aplicaciones la pantalla Splash. De lo contrario si estamos en MainActivity y pulsamos a ←, regresaría a Splash.

En el AndroidManifest.xml, tenemos que cambiar que la actividad lanzadora sea SplashActivity, a la cual además le ponemos un Theme sin ActionBar para que sea a pantalla completa, y la MainActivity la modificaremos para que se quede como normal.

```
<activity
   android:name=".SplashActivity"
   android:label="@string/app_name"
   android:theme="@style/Theme.MaterialComponents.DayNight.NoActionBar">
   ...
```

Opción 2

Recientemente muchos autores optan por otros métodos de splash rápido que no obliguen a crear una Activity a propósito. Es conocido como Launch Screen, para el cual no se necesita diseñar aparte otra Activity con su layout que haga de presentación. Lo vemos en el siguiente enlace.

- https://medium.com/@sneyderangulo/como-implementar-splash-screen-correctamente-en-android-f6abcc592b4c

6. ELEMENTOS MULTIMEDIA

6.1. Reproducción de sonido

Una de las cosas que más llaman la atención a las personas es la multimedia, imaginemos lo triste que sería un juego sin sonido, en aplicaciones normales, en ocasiones sería interesante hacer sonar cortas melodías, que deberían ser diferentes en caso de tener éxito o no, imaginemos hacer sonar unos aplausos si lo consigues o un cristal roto si fallas. Tenemos varias formas de reproducir sonidos, veremos seguidamente la clase **MediaPlayer**, la cual, es ideal por su facilidad.

Ejemplo de un método para reproducir sonido:

```
class SoundPlayer(private val context: Context) {

    fun reproducirSnd(snd: Int) {
        try {
            val mediaPlayer = MediaPlayer.create(context, snd)
            mediaPlayer?.apply {
                setVolume(0.5f, 0.5f)
                start()
                setOnCompletionListener {
                    // Liberar recursos después de la reproducción completa
                    release()
                }
            }
        } catch (exc: Exception) {
            // Manejar excepciones según sea necesario
            exc.printStackTrace()
        }
    }
}
```

Mediante setVolume(i, d) se fija el volumen izquierdo y derecho usando floats. Tal y como está el método reproducirá el sonido hasta terminar su tiempo.

Como vemos en la cabecera, se le debe de llamar con un entero como parámetro. Ese entero no es más que un recurso, recordemos que la clase R que se genera al compilar convierte todos los recursos en un número entero para hacer referencia a ellos. Por lo tanto, si tenemos en la carpeta **raw** un archivo de sonido mp3 llamado **misonido**, la llamada para hacerlo sonar sería así:

```
reproducirSnd(R.raw.misonido);
```

Por tanto, la llamada des de una actividad será así:

```
class MainActivity : AppCompatActivity() {

    private lateinit var soundPlayer: SoundPlayer

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        soundPlayer = SoundPlayer(this)

        // Llamada a la función para reproducir un sonido
        soundPlayer.reproducirSnd(R.raw.tu_sonido)
    }
}
```

Más información sobre MediaPlayer:

- https://developer.android.com/guide/topics/media/mediaplayer

6.2. Librería Picasso

Podemos obtener multitud de funcionalidades interesantes por medio de librerías, que nos ahorran un trabajo enorme. En el mundo Android la librería **Picasso** se ha convertido casi en un estándar, por su simplicidad y potencia. Mediante la misma, podemos descargar desde un servidor de internet y mediante su URL una imagen, e introducirla directamente dentro de un ImageView, todo ello con gestión de errores, y empleando una sola línea de código.

Antes tendremos que incorporar la dependencia en el dependencies de **gradle.build (:app)**.

```
implementation 'com.squareup.picasso:picasso:2.71828'
```

Además, requiere permiso de internet en el **AndroidManifest.xml**:

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

Y usarla en tan sencillo como:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Obtén una referencia al ImageView en tu diseño
        val imageView: ImageView = findViewById(R.id.imageView)

        // URL de la imagen que deseas cargar
        val imageUrl = "https://ejemplo.com/imagen.jpg"

        // Usa Picasso para cargar la imagen en el ImageView
        Picasso.get().load(imageUrl).into(imageView)
    }
}
```

Se puede apreciar que necesita de dos datos, la URL en un String, y una referencia al ImageView.

También admite transformaciones de tamaño de la imagen.

```
Picasso.get()
            .load(imageUrl)
            .resize(50, 50)
            .centerCrop()
            .into(imageView)
```

Por otro lado, podemos ver una imagen temporal mientras se carga, y otra si finalmente no puede hacerlo.

```
Picasso.get()
            .load(imageUrl)
            .placeholder(R.drawable.placeholder) // Recurso de imagen a mostrar
mientras se carga la imagen principal
            .error(R.drawable.error) // Recurso de imagen a mostrar si hay un error al
cargar la imagen principal
            .into(imageView)
```

También admite la carga de la imagen desde un recurso.

```
Picasso.get()
            .load(R.drawable.miimagen) // Recurso drawable de la imagen
            .into(imageView)
```

O más interesante, desde un archivo en el propio dispositivo.

```
// Ruta al archivo en el directorio "assets"
        val assetFilePath = "file:///android_asset/mifoto.png"

        // Usa Picasso para cargar la imagen desde el directorio "assets" en el
ImageView
        Picasso.get()
            .load(assetFilePath)
            .into(imageView)
```

Más información sobre Picasso:

- https://github.com/square/picasso

6.3. El vibrador

El **vibrador** es un recurso hardware, por lo tanto, tendremos que indicar que requerimos permiso para usarlo en el **AndroidManifest.xml**, añadiendo, por ejemplo, la siguiente línea dentro de la etiqueta <manifest> pero justo antes de la etiqueta <Aplication>.

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

Seguidamente, donde se desee usar el vibrador, ponemos en archivo Kotlin lo siguiente, es este caso vibrará cada vez que entre en la App (primero comprueba que está accesible el vibrador).

```
class MainActivity : AppCompatActivity() {

    private lateinit var vibrador: Vibrator

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
```

```
        vibrador = getSystemService(Context.VIBRATOR_SERVICE) as Vibrator

        if (!vibrador.hasVibrator()) {
            Toast.makeText(applicationContext, "No hay vibrador",
Toast.LENGTH_LONG).show()
        } else {
            vibrador.vibrate(1000)
        }
    }
}
```

Si se quisiera cancelar antes de hora sería mediante:

```
vibrador.cancel();
```

Es posible consultar desde Kotlin si tenemos permiso en el manifest así:

```
if (ContextCompat.checkSelfPermission(
                this,
                Manifest.permission.WRITE_CALENDAR
            ) != PackageManager.PERMISSION_GRANTED
        ) {
            // Sin permiso
        }
```

Más información sobre el vibrador:

- https://developer.android.com/reference/android/os/Vibrator

NOTA: Existen otros permisos como INTERNET, BLUETOOTH, ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, CALL_PHONE, READ_CONTACTS, WRITE_CONTACTS, SEND_SMS, RECEIVE_SMS, CAMERA, RECORD_AUDIO, WRITE_EXTERNAL_STORAGE, READ_EXTERNAL_STORAGE, READ_PHONE_NUMBERS, BODY_SENSORS, etc. que deberemos activar si lo requiere nuestra App.

Esquema de permiso en Android.

- http://www.androidcurso.com/index.php/recursos/41-unidad-7-seguridad-y-posicionamiento/282-el-esquema-de-permisos-en-android

7. SALIR DE LA ACTIVITY O DE LA APP

En los ejemplos que hemos elaborado no tenemos la opción de salir desde la propia aplicación, lo cual da sensación de poca profesionalidad.

Aplicación con solo una actividad

Para salir de nuestra aplicación, que solo tiene un layout, deberemos poner como respuesta a esos eventos en el sitio deseado una de las siguiente dos instrucciones:

```
// Retorna 0 al salir (resultado de ejecución correcta de la App)
System.exit(0)

// Cierra la actividad actual permitiendo ejecutar onDestroy(), el cual puede realizar tareas adicionales
finish()
```

En Android, sin embargo, la práctica común es no usar System.exit(0) para salir de la aplicación, ya que puede tener consecuencias inesperadas y no es la forma recomendada de manejar la finalización de una aplicación en Android. En su lugar, puedes simplemente finalizar la actividad actual: con finish().

Esto cierra la actividad actual y llama al método onDestroy(). La aplicación finalizará si no hay más actividades en el back stack. En términos generales, Android maneja la gestión del ciclo de vida de las aplicaciones y actividades, y es preferible seguir las prácticas recomendadas de Android para garantizar una experiencia de usuario más coherente y predecible.

Aplicación con diferentes actividades

No obstante, en una App con varias pantallas esto sólo funcionará si la usamos desde la activity principal, si lo usamos estando en otra pantalla regresaría, es decir, se saldría solo de la activity o layout actual, no de toda la aplicación. En estos casos, para cerrar el programa por completo, se usan otras formas, veamos ahora otros métodos que también podrían valer para salir.

```
val pid = android.os.Process.myPid()

// Mata el proceso llamando al sistema por medio del Pid
android.os.Process.killProcess(pid)

// Cierra todas las actividades presentes en la pila (~shutdown)
finishAffinity()
```

Ten en cuenta que cerrar la aplicación abruptamente puede no ser la mejor práctica en términos de experiencia del usuario, ya que podría causar pérdida de datos no guardados y no sigue las pautas de diseño de Android. En la mayoría de los casos, permitir que el usuario use el botón de retroceso o proporcionar un flujo de navegación adecuado es una práctica más recomendada. Sin embargo, puede haber situaciones específicas en las que necesites cerrar la aplicación por completo, y estos métodos pueden ser útiles en esos casos.