Capitulo 2

UD02. Interface de usuario. Ciclos de vida

Resultados de avaliación

**RA1.** Aplica tecnoloxías de desenvolvemento para dispositivos móbiles, e avalía as súas características e as súas capacidades.

**RA2.** Desenvolve aplicacións para dispositivos móbiles, para o que analiza e emprega as tecnoloxías e as librarías específicas.

Criterios de avaliación

* CA1.6 Describíronse os perfís que establecen a relación entre o dispositivo e a aplicación.
* CA1.7 Analizouse a estrutura de aplicacións existentes para dispositivos móbiles, e identificáronse as clases utilizadas.
* CA1.8 Realizáronse modificacións sobre aplicacións existentes.
* CA1.9 Utilizáronse emuladores para comprobar o funcionamento das aplicacións.
* CA2.1 Xerouse a estrutura de clases necesaria para a aplicación.
* CA2.2 Identificáronse as características das interfaces de usuario para dispositivos móbiles e técnicas específicas para o seu desenvolvemento e a súa adaptación.
* CA2.3 Analizáronse e utilizáronse as clases que modelan ventás, menús, alertas e controis para o desenvolvemento de aplicacións gráficas sinxelas.
* CA2.4 Utilizáronse as clases necesarias para a conexión e a comunicación con dispositivos sen fíos.
* CA2.5 Utilizáronse as clases necesarias para o intercambio de mensaxes de texto e multimedia.
* CA2.8 Realizáronse probas de interacción entre o usuario e a aplicación para mellorar as aplicacións desenvolvidas a partir de emuladores.
* CA2.9 Empaquetáronse e despregáronse as aplicacións desenvolvidas en dispositivos móbiles reais.
* CA2.10 Documentáronse os procesos necesarios para o desenvolvemento das aplicacións.

BC1. Análise de tecnoloxías para desenvolvemento de aplicacións en dispositivos móbiles.

* Estrutura dunha aplicación para dispositivo móbil.
* Modificación de aplicacións existentes.
* Uso do contorno de execución do administrador de aplicacións.
* Contornos integrados de traballo.
* Módulos para o desenvolvemento de aplicacións móbiles.
* Emuladores.
* Perfís: características, arquitectura e requisitos. Dispositivos soportados.
* Ciclo de vida dunha aplicación: descubrimento, instalación, execución, actualización e borrado.
* Ferramentas e fases de construción.
* Eventos da interface.
* Probas de interacción.
* Empaquetaxe e distribución.
* Documentación do desenvolvemento das aplicacións.
* Estrutura de clases dunha aplicación.
* Interfaces de usuario. Clases asociadas.
* Comunicacións: clases asociadas. Tipos de conexións

Capitulo 1

Linear Layout

LinearLayout es un tipo de diseño (layout) en Android que se utiliza para organizar elementos de interfaz de usuario en una vista en **línea recta**, ya sea horizontal o verticalmente. Es parte del sistema de diseño de Android y se utiliza para crear interfaces de usuario **simples** y **lineales**.

Un LinearLayout organiza sus elementos secuencialmente **uno después del otro** en el orden en que se agregan al diseño. Puedes controlar si los elementos se colocan horizontalmente o verticalmente utilizando la propiedad android:orientation en el archivo de diseño XML. Puedes establecer android:orientation en “horizontal” para que los elementos se alineen de izquierda a derecha o en “vertical” para que se alineen de arriba a abajo.

Aquí hay un ejemplo de un LinearLayout en XML con orientación vertical:

<LinearLayout

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<!-- Aquí puedes agregar elementos de interfaz de usuario, como botones, etiquetas, etc. -->

</LinearLayout>

Dentro de este LinearLayout, puedes agregar elementos como botones, etiquetas, campos de texto u otros elementos de interfaz de usuario que se organizarán verticalmente uno debajo del otro. Puedes personalizar las propiedades de diseño, como márgenes, pesos, alineación, etc., para controlar la apariencia exacta de los elementos en el diseño.

Crear proyecto

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
   * Selecciona “Start a new Android Studio project” en la pantalla de inicio.
   * Elige “Phone and Tablet” como tipo de dispositivo.
   * Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
   * En la siguiente pantalla, completa la información básica sobre el proyecto:
     + **Name**: Ingresa “UF1\_UD2\_1\_LinearLayout”.
     + **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
     + **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
     + **Language**: Selecciona “Kotlin” como lenguaje de programación.
     + **Minimum API level**: Selecciona “API 24: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
   * Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
   * Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad MainActivity.kt y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

Diseño

Elimanos todo el contenido del archivo activity\_main.xml, que define el diseño de nuestra aplicación, y generamos un nuevo LinearLayout con las siguientes configuraciones:

* Utilizamos autocompletado para el atributo android:xmlns.
* Utilizamos autocompletado para el atributo xmlns:tools.
* Establecemos el **ancho** y **alto** para que se ajusten a las dimensiones del contenedor principal.
* Seleccionamos una orientación **vertical**. Es importante tener en cuenta que si tuviéramos una orientación horizontal, los elementos se distribuirían de derecha a izquierda o de izquierda a derecha, dependiendo de la configuración del idioma. Por ejemplo, si el idioma por defecto fuera el árabe, los componentes se distribuirán de izquierda a derecha. Esto se puede especificar en el archivo AndroidManifest.xml con la propiedad:

android:supportsRtl="true"

Esta propiedad indica si admite Right to Left (es decir, una orientación horizontal de derecha a izquierda).

* Añadimos el **contexto** para asociar este layout con la actividad desde la que se va a lanzar. Este paso no es obligatorio, pero puede ser útil para determinadas herramientas. En la actividad, el diseño ya está asociado.
* Añadimos un **margen** de 16dp.

<?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"

android:padding="16dp">

# Componentes

## **Agregar Campos de Texto**

### **Campo de Texto Destinatario**

Para crear un campo de texto destinatario, seguiremos estas configuraciones:

* El **ancho** se ajusta automáticamente al tamaño del elemento padre, que a su vez está configurado para adaptarse al tamaño de la pantalla.
* La **altura** se ajustará dinámicamente según el contenido ingresado.
* Utilice el atributo android:hint para proporcionar una pista o mensaje de ejemplo en el campo de texto. Por ejemplo: android:hint="A:" (donde “hint” es una pista).
* Puede personalizar el formato de los contenidos según sea necesario, por ejemplo, para contraseñas, identificadores u otros tipos de datos, utilizando la propiedad inputType.

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="A:"

android:inputType="text"

/>

### GIF 1

### Caja de texto Asunto

Para crear un campo de texto para el “Asunto”, siga estas configuraciones:

* El ancho se ajusta automáticamente al tamaño del elemento padre, que a su vez está configurado para adaptarse al tamaño de la pantalla.
* La altura se ajustará dinámicamente según el contenido ingresado.
* Utilice el atributo android:hint para proporcionar una pista o mensaje de ejemplo en el campo de texto. Por ejemplo: android:hint="Mensaje" (donde “hint” es una pista).
* Si desea permitir la entrada de múltiples líneas de texto, configure la propiedad textMultiLine para admitir un formato de contenido diferente, como contraseñas, identificadores, etc.

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="Mensaje"

android:inputType="textMultiLine"

/>

Agregar un Botón

Para añadir un botón a la interfaz, siga estas configuraciones:

* El tamaño y la altura del botón se ajustarán automáticamente al contenido que contiene.
* Utilice el atributo android:text para establecer el texto del botón. Por ejemplo: android:text="Enviar".

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Enviar" />

Estilo de componentes

Una vez que hemos añadido los elementos básicos, es hora de darles formato:

1. **Agregar Márgenes**:

Para mejorar la apariencia visual, vamos a añadir márgenes tanto arriba como abajo a cada elemento. Esto se puede lograr con las siguientes líneas de código en las etiquetas de los elementos:

android:layout\_marginBottom="8dp"

android:layout\_marginTop="8dp"

1. **Aumentar el Tamaño de la Caja de Texto**:

Para hacer que la caja de texto sea más grande, vamos a utilizar la propiedad android:layout\_weight. Un valor igual o mayor que 1 distribuirá el espacio disponible entre los elementos. Puedes agregarlo así:

android:layout\_weight="1"

GIF 2

1. **Centrar el Contenido del Placeholder en la Caja de Texto**:

Podemos centrar el contenido dentro de la caja de texto utilizando la propiedad android:gravity. Esto asegurará que el texto dentro de la caja de texto esté alineado verticalmente en el centro. Puedes lograrlo de esta manera:

android:gravity="top"

También puedes combinar valores si necesitas centrar horizontalmente y verticalmente:

android:gravity="top|center"

1. **Posicionar el Botón**:

Para posicionar el botón a la derecha, puedes utilizar la propiedad android:layout\_gravity. Esto establecerá la ubicación del elemento dentro de su contenedor. Para colocar el botón a la derecha, puedes agregar lo siguiente:

android:layout\_gravity="right"

Además de todo esto, debemos cambiar los **literales** del texto.

# Práctica

En este ejercicio, crearemos una aplicación que muestra una lista de contactos en un LinearLayout vertical y permite a los usuarios interactuar con ellos.

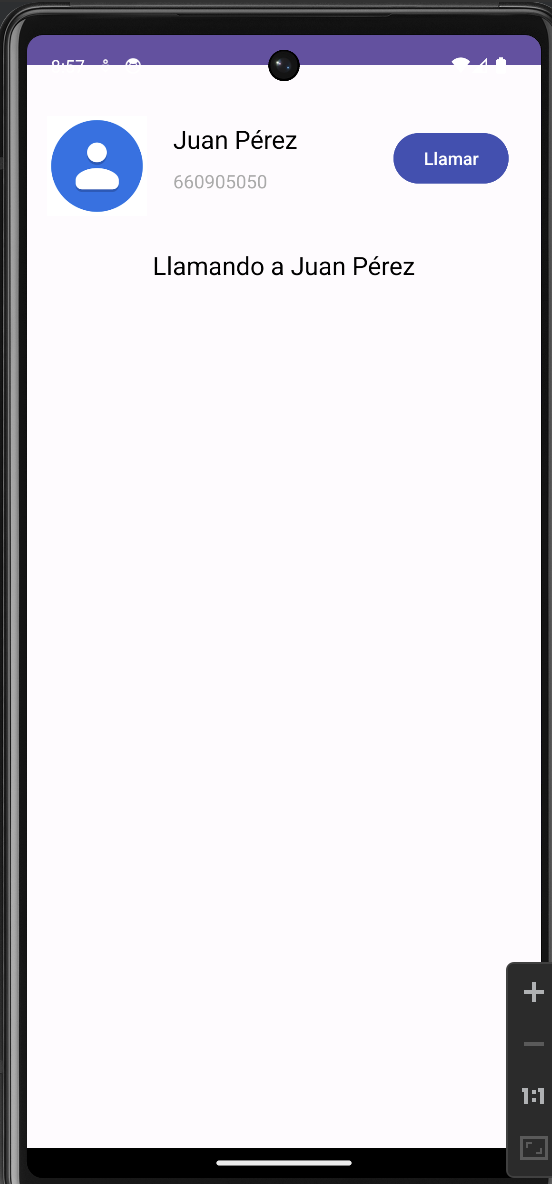
#### **Instrucciones**

1. Abre Android Studio y crea un nuevo proyecto de Android llamado “UDO2\_5\_Practica\_LinearLayout”.
2. En el archivo activity\_main.xml, crea una interfaz de usuario que contenga un LinearLayout vertical. Dentro de este LinearLayout, agregaremos elementos de vista para representar una lista de contactos.
3. Personaliza los elementos de vista con texto, imágenes y atributos de diseño según tus preferencias. Puedes usar elementos como ImageView [profile](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud02/1.linear_layout/4.practica/profile.png), TextView, y Button para representar un contacto. Para que la app se vea de la siguiente manera:



1. Implementa el código Java en el archivo MainActivity.java para que los elementos de vista puedan interactuar de la siguiente manera:

* Cuando se hace clic en el botón dentro del botón de “Llamar”, muestra un mensaje específico para ese contacto, como “Llamando a [nombre del contacto]”.



Capitulo 2

# Frame Layout

# Frame Layout

[FrameLayout](https://developer.android.com/reference/android/widget/FrameLayout) es uno de los tipos de contenedores de diseño (layouts) que se utilizan en el desarrollo de aplicaciones Android. Es un tipo de diseño que organiza los elementos secundarios (views) de forma superpuesta uno encima del otro, de modo que solo se muestra un elemento a la vez. Es decir, los elementos secundarios se apilan en capas y el último agregado se superpone a los anteriores.

Algunas características clave de FrameLayout incluyen:

* **Apilamiento de vistas**: Los elementos secundarios se colocan en la misma posición dentro del FrameLayout. Esto significa que ocupan la misma área y se superponen uno encima del otro. El último elemento agregado se muestra en la parte superior, ocultando los elementos anteriores.
* **Útil para superposición**: FrameLayout es especialmente útil cuando se necesita superponer vistas, como botones, imágenes o fragmentos, en una actividad o fragmento. Puedes usarlo para crear interfaces de usuario con capas.
* **Simplicidad**: Es uno de los layouts más simples y livianos disponibles en Android. No tiene reglas de alineación complejas, como LinearLayout o RelativeLayout.

## **Características Principales:**

* **Diseños Sencillos**: FrameLayout es ideal para diseñar interfaces de usuario simples y eficaces.
* **Apilamiento de Vistas**: Las vistas hijas se apilan unas sobre otras en función del orden en el que se añadieron. Esto permite controlar la superposición de elementos en pantalla.
* **Control de Posicionamiento**: Puedes controlar la posición de cada vista dentro del FrameLayout, lo que facilita la creación de diseños personalizados.

Crear proyecto

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
   * Selecciona “Start a new Android Studio project” en la pantalla de inicio.
   * Elige “Phone and Tablet” como tipo de dispositivo.
   * Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
   * En la siguiente pantalla, completa la información básica sobre el proyecto:
     + **Name**: Ingresa “UF1\_UD2\_2\_FrameLayout”.
     + **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
     + **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
     + **Language**: Selecciona “Kotlin” como lenguaje de programación.
     + **Minimum API level**: Selecciona “API 24: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
   * Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
   * Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad MainActivity.kt y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

# Diseño

Para lograr el resultado deseado, puedes reescribir las instrucciones de la siguiente manera:

#### **Reestructuración del Layout en activity\_main.xml**

Para configurar el diseño de la aplicación, primero eliminaremos todo el contenido del archivo activity\_main.xml. Luego, procederemos a crear un FrameLayout con las siguientes características:

* Utilizaremos la función de autocompletado para el atributo xmlns de Android.
* Asimismo, emplearemos el autocompletado para el atributo xmlns de la herramienta “tools”.
* Estableceremos el ancho y alto del diseño para que se ajusten a las dimensiones del contenedor principal.
* Elegiremos una orientación vertical. Es importante tener en cuenta que, en caso de una orientación horizontal, la disposición de los elementos variará de derecha a izquierda o de izquierda a derecha, según la configuración del idioma del dispositivo. Para gestionar esto, podemos definir la propiedad android:supportsRtl="true" en el archivo AndroidManifest.xml. Esta propiedad indica si la aplicación admite la orientación de derecha a izquierda (RTL).
* Agregaremos el contexto para asociar este diseño con la actividad desde la que se lanzará. Aunque este paso no es obligatorio, puede ser útil para ciertas herramientas.
* Por último, aplicaremos un margen de 16dp al diseño.

A continuación se muestra el código XML resultante:

<?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\_height="match\_parent"

android:layout\_width="match\_parent"

android:layout\_margin="16dp"

android:orientation="vertical"

tools:context=".MainActivity">

<!-- Aquí puedes agregar tus vistas y elementos de diseño -->

</FrameLayout>

### **Añadir imágenes**

Las tenemos que añadir en al carpeta de recursos. SEleccionamos la vista de ficheros del proyecto (veremos más recursos).

Arrastramos a la carpeta drawable. Si pinchamos con el botón derecho dentro de la imagen veremos que ya nos da la opción de convertirla a webp:

GIF 3

Para añadirla a nuestra app tenemos que crear un nuevo elemento, tenemos que establecer el ancho y el alto, lo dejamos al tamaño del recurso.

<ImageView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

Ahora tenemos que añadir el recurso, es decir, la imagen indicando la carpeta y el nombre de la imagen:

android:src="@drawable/iceberg"

Pero vamos a escalar la imagen:

* centreCrop: va a ocupar todo el contenido del layout. El escalado se hace uniforme.
* centerInside: lo mismo que en el anterior pero cuando uno de los coja los bordes pasa.
* center: La imagen es mucho más grande que el contenedor, al pone en el centro pero no la escala por lo que no se ve bien.

<ImageView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:src="@drawable/iceberg"

android:scaleType="centerCrop"/>

#### Multiresoluciones

Podemos tener diferentes resoluciones de nuestra imagen en función del dispositivo en el que se vaya a utilizar.

[Cómo brindar compatibilidad con diferentes densidades de píxeles](https://developer.android.com/training/multiscreen/screendensities?hl=es-419)

Hay que crear varias carpetas en la carpeta drawable para las dierentes imágenes con las diferentes resoluciones. En caso de que no tuviéramos

#### **Añadir la orta imagen (Ejercicio)**

Aparece encima de la otra imagen -> se apilan las distintas vistas en el orden en el que generamos código. Como tiene el fondo transparente podemos ver la imagen que hay abajo. la centramos y la bajamos:

android:layout\_gravity="center\_horizontal|bottom" />

### **Texto**

color: #RGB Se puede poner en hexadecimal Tamaño: android:textSize=“40sp” Estilo: android:textStyle=“bold”

# Frame Layout

Un [ScrollView](https://developer.android.com/reference/android/widget/ScrollView" \t "_blank) en Android es un contenedor que hereda de FrameLayout y se utiliza para permitir el desplazamiento de su contenido cuando no hay suficiente espacio para mostrarlo en la pantalla. Este contenedor solo admite un único elemento hijo y ofrece desplazamiento vertical. En lugar de agregar múltiples contenedores, podemos utilizar un solo ScrollView para lograr esta funcionalidad. Si necesitamos desplazamiento horizontal, podemos emplear un HorizontalScrollView.

La principal característica de un ScrollView es que permite a los usuarios desplazarse verticalmente (y en algunos casos horizontalmente) para acceder a todo el contenido que está fuera de la vista actual. Por ejemplo, si tienes un formulario largo o una lista de elementos que no caben completamente en la pantalla, puedes colocarlos dentro de un ScrollView para que los usuarios puedan desplazarse hacia arriba y abajo para interactuar con todo el contenido.

Crear proyecto

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
   * Selecciona “Start a new Android Studio project” en la pantalla de inicio.
   * Elige “Phone and Tablet” como tipo de dispositivo.
   * Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
   * En la siguiente pantalla, completa la información básica sobre el proyecto:
     + **Name**: Ingresa “UF1\_UD2\_3\_ScrollView”.
     + **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
     + **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
     + **Language**: Selecciona “Kotlin” como lenguaje de programación.
     + **Minimum API level**: Selecciona “API 24: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
   * Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
   * Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad MainActivity.kt y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

Diseño

Oara comenzar con nuestro proyecto, eliminamos todo el contenido del fichero activity\_main.xml que establece nuestro diseño de la aplicación y generaremos un ScrollView:

* Utilizaremos autocompletado para el atributo android **xmlns**.
* Utilizaremos autocompletado para el atributo **xmlns** de tools.
* Estableceremos el ancho y alto para que se ajusten a las **dimensiones** del contenedor principal.
* Añadimos el contexto para asociar este layout con la actividad desde la que se va a lanzar.

<?xml version="1.0" encoding="utf-8"?>

<ScrollView

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"

tools:context=".MainActivity"

>

</ScrollView>

Como sabemos, este elemento sólo podemos tener un único componente, pero nosotros queremos tener dos: un **elemento de texto** y un **botón**. Por lo que, añadimos un LinearLayout con las siguientes características:

* Estableceremos el ancho y alto para que se ajusten a las **dimensiones** del contenedor principal.
* Añadir un margen de 16dp.
* Seleccionaremos una orientación **vertical**.

Dentro de este Layout crearemos los elementos que necesitamos:

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical"

android:padding="16dp"

>

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="Mensaje"

android:inputType="textMultiLine" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Enviar" />

</LinearLayout>

Vídeo

Si eliminamos el ScrollView y dejamos solo el LinearLayout como elemento raíz, notaríamos que no podemos realizar un desplazamiento vertical. Aunque podríamos desplazarnos dentro de la caja de texto, no tendríamos la capacidad de visualizar el botón que está fuera de la vista.

# Imágenes

Para agregar imágenes a tu aplicación, es necesario seguir algunos pasos importantes. A continuación, se describe el proceso detalladamente:

**1. Organización en la Carpeta de Recursos:**

Primero, debes asegurarte de que tus imágenes estén ubicadas en la carpeta de recursos de tu proyecto. Para ello, realiza los siguientes pasos:

* Abre la vista de **ficheros** de tu proyecto (donde puedes ver la estructura de archivos).
* Navega hasta la carpeta **“drawable”** dentro de la carpeta de recursos. Si no existe, créala.

**2. Conversión a Formato WebP (Opcional):**

Si deseas optimizar tus imágenes para mejorar la eficiencia, puedes convertirlas al formato WebP. Esto se puede hacer fácilmente:

* En la carpeta “drawable”, haz clic derecho en la imagen que deseas convertir.
* Verás una opción para convertirla a formato WebP.

**3. Agregar una ImageView:**

Ahora, debes incorporar la imagen en tu aplicación utilizando una ImageView. Define la ImageView en el archivo XML de diseño de tu actividad de la siguiente manera:

<ImageView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:src="@drawable/iceberg"

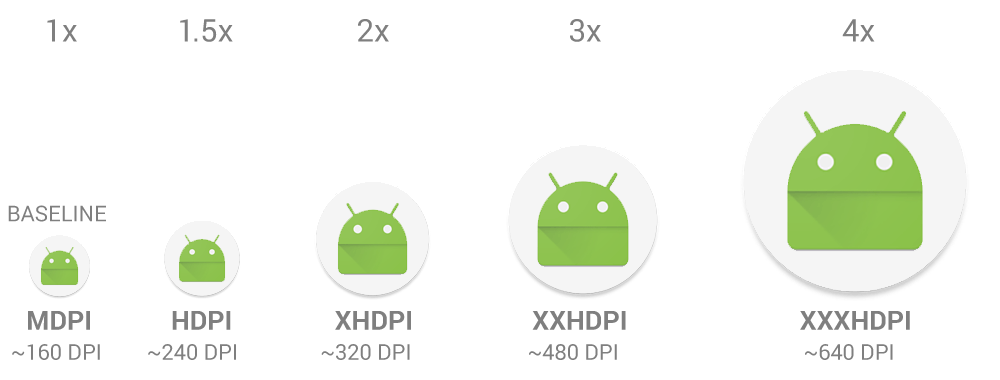
android:scaleType="centerCrop"/>

El código anterior especifica las siguientes características:

* android:layout\_width y android:layout\_height establecen el ancho y alto de la ImageView según las dimensiones de tu recurso de imagen.
* android:src especifica la imagen que deseas mostrar, haciendo referencia al nombre de la imagen en la carpeta “drawable”.
* android:scaleType define cómo se escalará la imagen dentro de la ImageView. En este caso, se utiliza “centerCrop” para llenar todo el contenido del diseño, manteniendo la relación de aspecto y recortando los bordes si es necesario.

#### **Densidades de pantalla**

Podemos tener diferentes [resoluciones](https://developer.android.com/training/multiscreen/screendensities?hl=es-419) de nuestra imagen en función del dispositivo en el que se vaya a utilizar.



Android proporciona carpetas de recursos específicas para diferentes densidades de pantalla, como drawable-mdpi, drawable-hdpi, drawable-xhdpi, drawable-xxhdpi, y más. Puedes colocar imágenes y recursos gráficos optimizados para cada densidad en estas carpetas. Android seleccionará automáticamente los recursos adecuados según la densidad de pantalla del dispositivo.

#### **Añadir la orta imagen (Ejercicio)**

Añade la otra imagen a continuación de la anterior.

<ImageView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:src="@drawable/tux"

android:layout\_gravity="center\_horizontal|bottom" />

Las distintas vistas se **superponen**, apareciendo una encima de la otra en el orden en que se generó el código. Dado que estas vistas tienen un fondo **transparente**, permiten que la imagen subyacente sea visible. Para centrar la vista y moverla hacia abajo, utilizamos la propiedad android:layout\_gravity de la siguiente manera:

android:layout\_gravity="center\_horizontal|bottom"

Esta configuración posiciona la vista en el centro horizontal de su contenedor y la alinea en la parte inferior del mismo.

# Texto

Por último nos quedaría añadir el texto de nuestra aplicación, vamos a definir un TextView con el texto “Tux Loves Linux!” que se muestra en negritas, con un tamaño de texto grande. El TextView se coloca en la parte superior y se centra horizontalmente dentro de su contenedor, y el color del texto se define como blanco (asumiendo que haya un recurso de color “white” definido en el proyecto).

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Tux Loves Linux!"

android:layout\_gravity="center\_horizontal|top"

android:textColor="@color/white"

android:textSize="40sp"

android:textStyle="bold

# Práctica

En este diseño, crearemos un diseño de pila de tarjetas en el FrameLayout. Cada tarjeta contendrá una imagen y un título, y las tarjetas se apilarán una encima de la otra.

#### **Instrucciones**

1. Abre Android Studio y crea un nuevo proyecto de Android llamado “UDO2\_6\_Practica\_FrameLayout”.
2. En el archivo activity\_main.xml, crea una interfaz de usuario que contenga un FrameLayout.
3. Personaliza los elementos de vista con texto, imágenes y atributos de diseño según tus preferencias. Puedes usar elementos como ImageView y TextView pra que la app se vea de la siguiente manera:



Fotos:

* [Ourense](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud02/2.frame_layout/6.practica/ourense.jpg)
* [Santiago](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud02/2.frame_layout/6.practica/santiago.jpg)

1. Implementa el código Java en el archivo MainActivity.java para que los elementos de vista puedan interactuar de la siguiente manera:

* Cuando se hace clic en la imagen, muestra un mensaje específico para esa ciudad.
* Cuando se hace clic en la imagen, se escucha un sonido.

# Constrait Layout

Los **[Constraint Layouts](https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout" \t "_blank)** son una característica importante en el desarrollo de aplicaciones Android. Estos diseños son el enfoque **predeterminado** en Android Studio y se han convertido en un componente esencial para crear interfaces de usuario flexibles y eficientes.

Está disponible como una librería de soporte que puede utilizarse desde el nivel de API 9, lo que significa que son compatibles con una amplia variedad de dispositivos Android. Los Constraint Layouts son especialmente valiosos para **evitar jerarquías** complejas de vistas, lo que a su vez mejora el rendimiento de la aplicación.

Una de las características distintivas de Constraint Layout es su capacidad para definir **restricciones** que controlan la **posición** y el **tamaño** de los componentes dentro de él. Estas restricciones pueden abordar márgenes, posicionamiento relativo y dimensionamiento, lo que proporciona un control preciso sobre la disposición de los elementos de la interfaz de usuario.

Puedes encontrar una guía detallada sobre cómo utilizar Constraint Layout en la [documentación oficial de Android](https://developer.android.com/training/constraint-layout?hl=es-419), que te ayudará a dominar esta potente herramienta de diseño.

Constraint Layout es solo una parte del conjunto de herramientas conocido como [Android Jetpack](https://developer.android.com/jetpack?hl=es-419). El objetivo de Jetpack es acelerar el desarrollo de aplicaciones al tiempo que se siguen las mejores prácticas. Proporciona bibliotecas para la arquitectura, manejo de datos, navegación y diseño de interfaces de usuario, entre otras cosas.

En resumen, Constraint Layouts son esenciales para el desarrollo de interfaces de usuario **flexibles** en Android, y forman parte de las herramientas poderosas que Android Jetpack ofrece a los desarrolladores para crear aplicaciones eficientes y de alta calidad.



Todas las actividades que podemos hacer con esas librerías se pueden organizar en cuatro grandes grupos:

* Arquitectura.
* Datos.
* Navegación entre las pantallas.
* Interfaz de usuario; layouts, Emojis, etc.

Otro objetivo es que el código funcione con independencia del código que se está ejecutando. Entre las muchas tareas que nos proporciona vamos a empezar a ver el ConstraitLayout.

Crear proyecto

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
   * Selecciona “Start a new Android Studio project” en la pantalla de inicio.
   * Elige “Phone and Tablet” como tipo de dispositivo.
   * Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
   * En la siguiente pantalla, completa la información básica sobre el proyecto:
     + **Name**: Ingresa “UF1\_UD2\_4\_ConstraitLayout”.
     + **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
     + **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
     + **Language**: Selecciona “Kotlin” como lenguaje de programación.
     + **Minimum API level**: Selecciona “API 24: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
   * Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
   * Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad MainActivity.kt y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

# Dependencias

Dentro de la carpeta “app”, encontramos un archivo llamado build.gradle, que es un script de configuración fundamental para nuestro proyecto Android. En este archivo, definimos las bibliotecas y dependencias que nuestro proyecto utilizará. Entre estas dependencias, podemos observar la presencia de la biblioteca ConstraintLayout. Si esta biblioteca no está presente en nuestro proyecto en el momento de la construcción, se descargará automáticamente.

El archivo build.gradle se ve así:

dependencies {

implementation("androidx.core:core-ktx:1.9.0")

implementation("androidx.appcompat:appcompat:1.6.1")

implementation("com.google.android.material:material:1.9.0")

implementation("androidx.constraintlayout:constraintlayout:2.1.4")

testImplementation("junit:junit:4.13.2")

androidTestImplementation("androidx.test.ext:junit:1.1.5")

androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")

}

Las dependencias, como ConstraintLayout, se agregan mediante la función implementation. Esto significa que estamos incluyendo esas bibliotecas en nuestro proyecto.

Para asegurarnos de que las bibliotecas se descarguen correctamente, especificamos los repositorios desde los cuales se obtendrán estas dependencias en el archivo settings.gradle:

dependencyResolutionManagement {

repositoriesMode.set(RepositoriesMode.FAIL\_ON\_PROJECT\_REPOS)

repositories {

google()

mavenCentral()

}

}

En este caso, estamos utilizando dos repositorios principales: “google” y “mavenCentral”. Gracias a esta configuración, el sistema descargará las bibliotecas necesarias desde estos repositorios automáticamente cuando construyamos el proyecto.

En resumen, el archivo build.gradle es esencial para gestionar las dependencias de nuestro proyecto Android, y la biblioteca ConstraintLayout se encuentra entre las dependencias que hemos configurado. La configuración de repositorios asegura que estas bibliotecas se descarguen correctamente sin necesidad de una intervención adicional.

# Diseño

En el archivo activity\_main.xml, procedemos a eliminar el elemento TextView, dejando el archivo de la siguiente manera:

<?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"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

tools:context=".MainActivity">

</androidx.constraintlayout.widget.ConstraintLayout>

A continuación, en la vista de diseño, agregamos un botón. Vemos que lo podemos arrastrar y mover a cualquier lado de la pantalla, pero es importante destacar que arrastrarlo y posicionarlo libremente en la pantalla no es una práctica adecuada, ya que la posición del botón puede variar según la orientación del dispositivo.

GIF 4

En lugar de eso, vamos a utilizar **restricciones** (constraints) para posicionar el botón de manera precisa. Si observamos el botón, notamos que tiene unos accionadores relacionados con el tamaño del botón, y cualquier cambio que realicemos aquí se reflejará en el código.

GIF 5

Además, tenemos accionadores **circulares** que nos permiten ajustar el comportamiento de la vista. Por ejemplo, si arrastramos el botón hacia un lado de la pantalla, aparecerá una nueva **restricción** que indica que el “inicio” del botón está conectado al “inicio” del padre. Si cambiamos la orientación del dispositivo, veremos que el botón se mantiene ajustado al lado restringido.

GIF 6

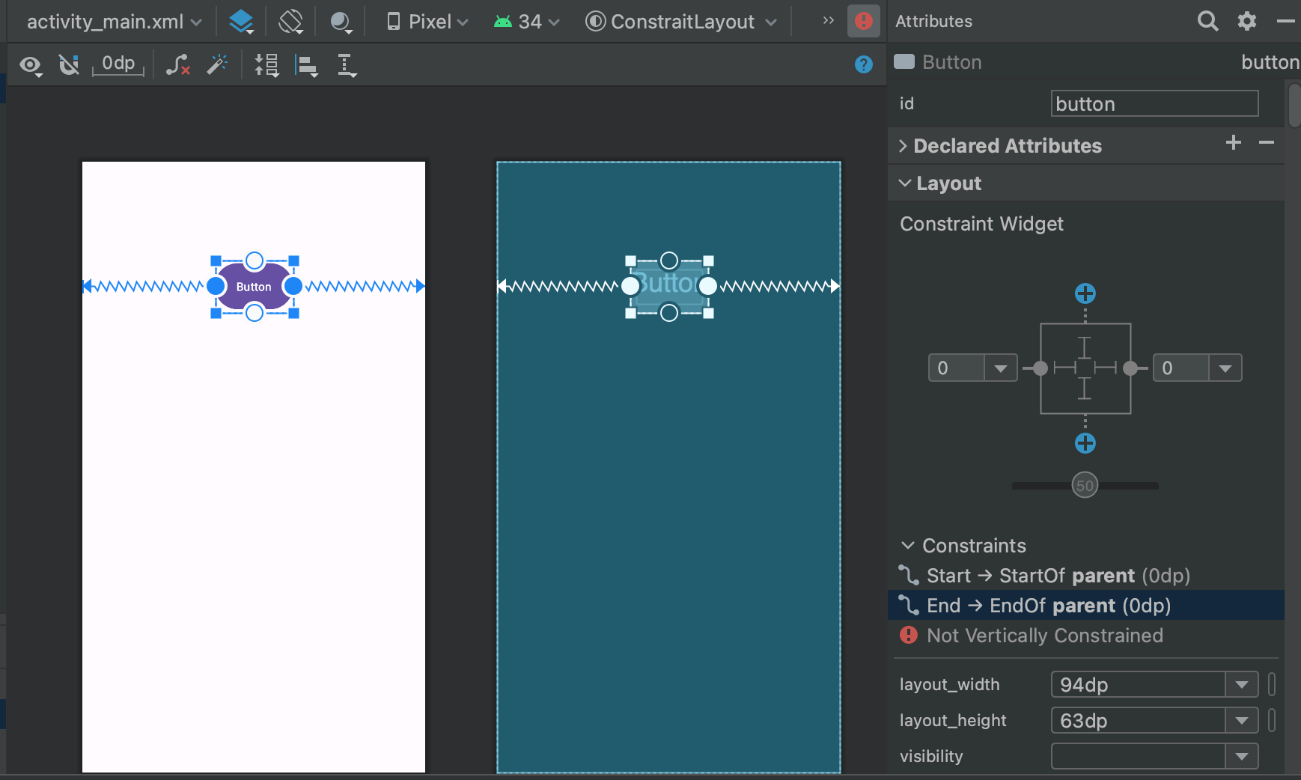
Si ahora ajustamos el lado opuesto a la otra parte de la pantalla, veremos cómo el elemento se posiciona en el **centro**. Podemos pensar en estas restricciones como “muelles” que están tirando del componente en ambas direcciones.

En el panel de la derecha, al seleccionar un elemento, encontramos un **widget** que nos permite **revisar** las restricciones aplicadas. Además, podemos ajustar el peso de estas restricciones, tanto horizontal como verticalmente, para controlar la disposición precisa del componente en la vista.

Si ahora ajustamos el lado contrario al otro lado de la pantalla vemos como el elemento se posiciona en medio, podemos comprobar como estas restricciones funcionan como “**muelles**” por decirlo de alguna manera que están tirando del componente a ambos lados.

GIF 7

En el panel de la derecha, al seleccionar un elemento, tenemos un widget donde podemos comprobar las restricciones que tenemos:



En este componente tenemos un elemento que nos permite controlar el peso que tiene esas restricciones, tanto horizontalmente como verticalmente.

GIF 8

# Vistas

## **Dimensiones y Márgenes de las Vistas**

Además de aplicar restricciones de posición, también podemos aplicar restricciones al tamaño de un elemento en nuestra interfaz. En las propiedades, tenemos la capacidad de definir diferentes restricciones. Por ejemplo, si establecemos el ancho de un botón en 0dp, veremos cómo automáticamente se expande para ocupar todo el espacio definido por esa restricción, que equivale al ancho de la pantalla.

GIF 9

Esta acción es similar a establecer un valor de “match\_parent” en el atributo de ancho del botón en el archivo XML:

<Button

android:id="@+id/button"

android:layout\_width="match\_parent"

...

Aunque el resultado visual es el mismo en ambos casos, existe una pequeña **diferencia** conceptual, **“match\_parent”** se refiere al ancho del contenedor padre, mientras que cuando ajustamos el tamaño en el ancho mediante una **restricción**, en realidad estamos configurando el tamaño en relación con esa restricción específica.

En las cajas de restricciones, se establecen números que son **potencias de dos**. Estos números permiten definir márgenes en relación con la restricción en sí, lo que proporciona un mayor control sobre la disposición de los elementos en la interfaz.

En resumen, además de posicionar elementos en la interfaz, ***ConstraintLayout*** nos permite controlar con **precisión** el **tamaño** y los **márgenes** de las vistas mediante la configuración de restricciones específicas. Esto ofrece una mayor flexibilidad en la creación de interfaces de usuario personalizadas y responsivas.

GIF 10

## **Alineación entre vistas**

Vamos a incorporar otro botón y establecer una restricción que conecte la parte inferior del primer botón con la parte superior del segundo botón:

GIF 11

En este escenario, estamos definiendo una **restricción** en la que la **posición** de un botón depende de la posición del otro. Cuando movemos el primer botón, notamos que el segundo botón también se **desplaza**, y esto se debe a la restricción que acabamos de crear:

GIF 12

Por lo general, deseamos que los elementos de nuestra aplicación estén alineados de alguna manera para mantener una **apariencia uniforme** en nuestro diseño. Una forma sencilla de lograrlo es seleccionar ambos botones, lo que activará una opción en el menú que nos permitirá explorar diversas alternativas de **alineación**. Al seleccionar una de estas opciones, se generará automáticamente una restricción:

De esta manera, el primer botón tendrá dos restricciones asociadas:

1. La distancia entre ambos botones.
2. La alineación de los bordes izquierdos de ambos botones.

Con estas restricciones en su lugar, si movemos el primer botón, observamos cómo el segundo botón se ajusta automáticamente para mantener ambas restricciones:

GIF 13

Todas estas restricciones también pueden configurarse de manera **manual** seleccionando el borde izquierdo de un botón y conectándolo con el borde izquierdo del otro botón según sea necesario.

# Guías

## **Uso de Guías**

Otra técnica útil para alinear componentes es el uso de **guías**, una práctica común en programas de edición de imágenes como Photoshop o Gimp. En ConstraintLayout, puedes utilizar guías para lograr alineaciones precisas de la siguiente manera:

1. Dirígete al menú desplegable “Guidelines” y selecciona la opción para agregar una guía vertical.
2. Una vez que hayas añadido la guía, puedes ajustar los elementos de la interfaz para que se alineen con ella. Cuando mueves la guía, notarás que los elementos también se desplazan verticalmente junto con ella:

GIF 14

1. Al seleccionar la guía, aparecerá un número en la pantalla. Si haces clic en la flecha en la parte inferior de la guía, podrás alternar entre un valor en porcentaje y un valor en píxeles (dp):

GIF 15

Esto significa que puedes ajustar la distancia de la guía hacia uno de sus extremos (izquierda o derecha) o en un porcentaje de la pantalla. Es importante tener en cuenta que el uso de porcentajes puede ser más adecuado si deseas que tu diseño sea adaptable a diferentes tamaños de pantalla, mientras que los valores fijos en píxeles pueden generar diferencias en la disposición dependiendo del dispositivo.

Crea una caja de texto y un botón debajo de ella en ConstraintLayout. Observa cómo el botón se desplaza automáticamente a medida que escribes en la caja de texto. Este comportamiento se logra automáticamente gracias a las restricciones y la alineación de elementos en la interfaz mediante ConstraintLayout.

## **Guías y Barreras**

En el caso de que necesitemos algo más flexible o una guía que se modifiquen durante la ejecución de la aplicación y que vaya marcando el posicionamiento de los elementos. Hablamos de las guías móviles. Para poder ver este término, elminamos los elementos que tenemos en nuestra aplicación y vamos a añadir dos cajas de texto multilínea de forma que vaya creciendo a medida que se vaya escribiendo. Debajo de las cajas habrá un botón que se debe ir desplazando a medida que las cajas vayan creciendo.

Vamos a empezar creando una guía vertical y la situamos en el medio de la pantalla utilizando un porcentaje del 50%. Lo siguiente será añadir esas cajas de texto editables (Text Multiline), ajustamos su posición añadiendo una restricción con la parte superior, izquierda y derecha de la pantalla añadiendo un margen de 24dp, ajustamos la altura del botón a su contenido:

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

Ahora añadimos un botón, lo ajustamos tanto vertical como horizontalmente y lo colocamos en algún punto de la pantalla:

GIF 16

Si ahora ejecutamos nuestra aplicación vemos que si las cajas de texto editables crecen mucho nuestro botón se queda en la misma posición:

GIF 17

Lo que necesitamos es que ese botón se desplazara, ajustara su posición, a medida que el resto de elementos cambien de posición.

### **Barreras móviles**

La diferencia de las barreras y las guías es que las guías son fijas y las barreras pueden desplazarse a medida que haya algún cambio en la aplicación.

* **Añadir una barrera móvil**

Desde el desplegable de guías (barra Design Tools > Guidelines), pulsar sobre Horizontal Barrier

* **Añadir a la barrera una restricción que referencie las cajas de texto**

El paso anterior nos habrá añadido una nueva barrera horizontal (podemos comprobarlo en el Component Tree donde debería aparecer algo como barrierN). Ahora, debemos añadirle una restricción para que su posición se ajuste en función de la posición de las cajas de texto editables. Para ello, desde el panel Component Tree, seleccionamos las dos cajas de texto y las arrastramos para colocarlas como hijos del componente barrera.

* **Cambiar la restricción a la parte inferior de las cajas de texto**

Por defecto, la nueva restricción se conecta con la parte superior de las cajas de texto. Para modificarlo, de forma que la restricción de la barrera esté conectada con la parte inferior de las cajas de texto (de forma que ésta se desplace al variar el tamaño de las mismas), tenemos que modificar su propiedad barrierDirection estableciendo su valor a bottom.

GIF 18

Ahora ya tenemos una barrera móvil que se irá desplazando a medida que los elementos superiores crezcan. Lo siguiente que tendremos que hacer será añadir una restricción para que lo haga también el botón. Para ello, seleccionamos el botón y buscamos la restricción layout\_constraintTop\_toTopOf al que le asignamos el id de la barrera: @id/barrier8:

Si ahora ejecutamos nuestra aplicación, vemos que el botón se va desplazando a medida que aumentamos el campo de texto.

GIF 19

## **Cadenas**

Son un grupos de vistas que están conectadas entre ellas utilizando las restricciones, normalmente se utilizan para mantener el mismo espaciado entre ellas con independencia de la horientación, para que se distribuya uniformemente el espacio sobrante entre los distintos elementos. Para ver las cadenas vamos a partir de un espacio en blanco, por lo que borramos todo lo que tenemos hasta ahora y añadimos tres botones y añadimos las siguientes restricciones:

* Al primer botón le añadimos una restricción con la parte superior de la pantalla dando un margen de 64.
* Añadimos una restricción a los otros dos botones para que se alineen en la parte superior con el botón actual.

Ahora los tenemos alineados horizontalmente, lo que queremos añadir es un espaciado común entre los tres botones , que no cambie aunque se cambie la posición del dispositivo. Para ello tenemos que hacer clic en el botón derecho en el primer botón y en el menú de “chain” crear una cadena horizontal.

# Práctica

En este ejercicio, crearemos una pantalla de inicio de sesión simple con dos campos de entrada (nombre de usuario y contraseña) y un botón de inicio de sesión. Usaremos ConstraintLayout para crear una interfaz de usuario flexible y responsiva.

#### **Instrucciones**

1. Abre Android Studio y crea un nuevo proyecto de Android llamado “UDO2\_7\_Practica\_ConstraintLayout”.
2. En el archivo activity\_main.xml, crea una interfaz de usuario que contenga un ConstraintLayout .
3. Personaliza los elementos de vista con texto, imágenes y atributos de diseño según tus preferencias (ambos campos de texto son multilínea) Para que la app se vea de la siguiente manera:



1. Implementa el código Java en el archivo MainActivity.java para que los elementos de vista puedan interactuar de la siguiente manera:

* Al hacer clic en el botón “Iniciar Sesión”, se mostrará un mensaje de prueba con los valores ingresados en los campos de entrada.
* Ponle un scroll para que no se pierda el botón.



Capitulo 3

# UD03. Fragmentos y navegación

### **Resultados de avaliación**

**RA1.** Aplica tecnoloxías de desenvolvemento para dispositivos móbiles, e avalía as súas características e as súas capacidades.

**RA2.** Desenvolve aplicacións para dispositivos móbiles, para o que analiza e emprega as tecnoloxías e as librarías específicas.

### **Criterios de avaliación**

* CA1.6 Describíronse os perfís que establecen a relación entre o dispositivo e a aplicación.
* CA1.7 Analizouse a estrutura de aplicacións existentes para dispositivos móbiles, e identificáronse as clases utilizadas.
* CA1.8 Realizáronse modificacións sobre aplicacións existentes.
* CA1.9 Utilizáronse emuladores para comprobar o funcionamento das aplicacións.
* CA2.1 Xerouse a estrutura de clases necesaria para a aplicación.
* CA2.4 Utilizáronse as clases necesarias para a conexión e a comunicación con dispositivos sen fíos.
* CA2.5 Utilizáronse as clases necesarias para o intercambio de mensaxes de texto e multimedia.
* CA2.8 Realizáronse probas de interacción entre o usuario e a aplicación para mellorar as aplicacións desenvolvidas a partir de emuladores.
* CA2.9 Empaquetáronse e despregáronse as aplicacións desenvolvidas en dispositivos móbiles reais.
* CA2.10 Documentáronse os procesos necesarios para o desenvolvemento das aplicacións

### **BC1. Análise de tecnoloxías para desenvolvemento de aplicacións en dispositivos móbiles.**

* Estrutura dunha aplicación para dispositivo móbil.
* Modificación de aplicacións existentes.
* Uso do contorno de execución do administrador de aplicacións.
* Contornos integrados de traballo.
* Módulos para o desenvolvemento de aplicacións móbiles.
* Emuladores.
* Perfís: características, arquitectura e requisitos. Dispositivos soportados.
* Ciclo de vida dunha aplicación: descubrimento, instalación, execución, actualización e borrado.
* Eventos da interface.
* Descubrimento de servizos
* Empaquetaxe e distribución.
* Documentación do desenvolvemento das aplicacións.
* Contexto gráfico. Imaxes.
* Comunicacións: clases asociadas. Tipos de conexións

Capitulo 1

Fragmentos

Un fragmento en Android es una porción **modular** y reutilizable de la interfaz de usuario de una aplicación. Puede considerarse como un “subcomponente” de una actividad, y su principal propósito es representar una parte específica de la interfaz de usuario o una funcionalidad de la aplicación dentro de una pantalla. Los fragmentos son utilizados principalmente para crear interfaces de usuario flexibles y adaptables en dispositivos con diferentes tamaños de pantalla, como teléfonos y tabletas.

Aquí hay algunas características clave de los fragmentos en Android:

1. **Reutilización:** Los fragmentos pueden ser reutilizados en diferentes partes de la aplicación y en múltiples actividades. Esto permite un desarrollo más modular y eficiente.
2. **Ciclo de vida propio:** Los fragmentos tienen su propio ciclo de vida, similar al ciclo de vida de una actividad. Esto significa que pueden responder a eventos y gestionar su propio estado.
3. **Interacción:** Los fragmentos pueden comunicarse entre sí y con la actividad que los contiene a través de la actividad anfitriona. Esto facilita la construcción de interfaces de usuario complejas con componentes interactivos.
4. **Diseño adaptable:** Los fragmentos se utilizan comúnmente en aplicaciones que necesitan adaptarse a diferentes tamaños de pantalla. Puedes cargar y organizar fragmentos de manera dinámica según el espacio disponible en la pantalla.
5. **Transacciones de fragmentos:** Los fragmentos se agregan, reemplazan o eliminan dinámicamente en tiempo de ejecución utilizando transacciones de fragmentos, lo que permite una gestión flexible de la interfaz de usuario.
6. **Compatibilidad con back stack:** Los fragmentos pueden ser gestionados en una pila de retroceso (back stack), lo que permite una navegación fluida hacia atrás y hacia adelante dentro de una actividad.

En resumen, un fragmento en Android es una herramienta poderosa que permite dividir y organizar la interfaz de usuario y la funcionalidad de una aplicación en componentes más pequeños y reutilizables, lo que facilita el desarrollo de aplicaciones flexibles y adaptables a una variedad de dispositivos y escenarios.

Crear proyecto

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
   * Selecciona “Start a new Android Studio project” en la pantalla de inicio.
   * Elige “Phone and Tablet” como tipo de dispositivo.
   * Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
   * En la siguiente pantalla, completa la información básica sobre el proyecto:
     + **Name**: Ingresa “UD03\_1\_Fragmentos”.
     + **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
     + **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
     + **Language**: Selecciona “Kotlin” como lenguaje de programación.
     + **Minimum API level**: Selecciona “API 35: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
   * Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
   * Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad MainActivity.kt y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

# Recursos

Crea los diferentes recursos para los distintos textos que aparecen en nuestra aplicación, tanto en inglés como en castellano:

### **Inglés**

<?xml version="1.0" encoding="utf-8"?>

<resources>

<string name="app\_name">UD1\_UD03\_SecretApp</string>

<string name="welcome\_text">Welcome to the Secret Message app! Use this app to encrypt a secret message.</string>

<string name="start">Start</string>

<string name="next">Next</string>

<string name="encrypt\_text">Here is your encrypted message!</string>

<string name="message\_text">Please, enter your message</string>

</resources>

### **Castellano**

<resources>

<string name="app\_name">UD1\_UD03\_SecretApp</string>

<string name="welcome\_text">¡Bienvenido a la aplicación Secret Message! Usa esta app para encriptar un mensaje secreto.</string>

<string name="start">Inicio</string>

<string name="next">Siguiente</string>

<string name="encrypt\_text">¡Aquí está tu mensaje encriptado!</string>

<string name="message\_text">Por favor, introduce tu mensaje</string>

</resources>

Crear Fragmento

Para crear un fragmento, se deben seguir estos pasos:

1. Ve al menú “File” y selecciona “Fragment”.
2. En esta sección, encontrarás varias opciones. Algunos fragmentos ya están predefinidos y tienen comportamientos específicos. En este caso, seleccionaremos la opción en **blanco**.
3. Aparecerá un asistente en el que deberás especificar el nombre del fragmento. En este ejemplo, utilizaremos “WelcomeFragment”. Automáticamente, se generará el nombre del archivo XML para el diseño del fragmento (siguiendo la convención de invertir el nombre y usar guiones bajos, similar a cómo se hace en los layouts).
4. Haz clic en “Finalizar” y verifica que se haya creado un archivo Kotlin y un archivo XML asociados a nuestro fragmento, de manera similar a lo que sucede al crear la actividad principal.

GIF 20

# Función onCreateView

Dirígete al archivo Kotlin correspondiente y elimina el código generado que, por el momento, no es necesario. Mantendremos únicamente lo esencial para este fragmento:

class WelcomeFragment : Fragment() {

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

// Inflamos el diseño de este fragmento

return inflater.inflate(R.layout.fragment\_welcome, container, false)

}

}

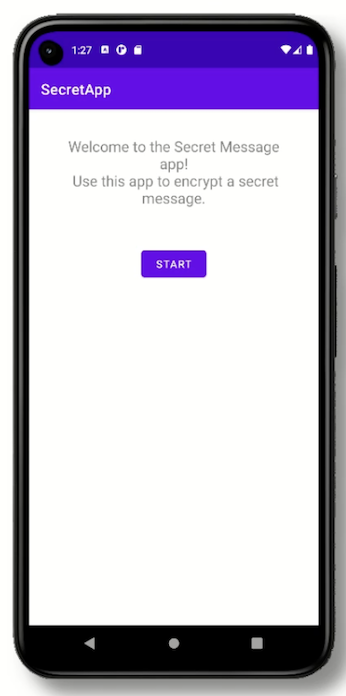
Este código define la clase WelcomeFragment, que hereda de la clase Fragment, de manera similar a lo que hicimos en nuestro MainActivity. Al igual que teníamos un método onCreate, aquí también tenemos un método que se sobrescribe y se ejecuta en el momento en que se lanza este fragmento.

En este método, utilizamos el objeto inflater para inflar (crear) la interfaz de usuario asociada a este fragmento a partir del archivo XML que se generó simultáneamente con este archivo. Este método devuelve una **vista** que representa la interfaz de usuario del fragmento.

Los fragmentos son componentes adicionales en nuestra interfaz de usuario, y este código se encarga de inflar la interfaz de usuario definida en el archivo XML asociado al fragmento.

# Fragment Welcome

Vamos a diseñar el primer fragmento que constituirá la pantalla inicial. Queremos lograr la siguiente apariencia:



Para construir esta pantalla, podemos utilizar un LinearLayout con las siguientes especificaciones:

**Fragmento: Welcome**

* **Diseño:**
  + Centrado horizontalmente.
* **TextView:**
  + Centrado horizontalmente.
  + Margen de 20dp.
  + Tamaño de texto de 20sp.
* **Botón:**
  + Margen superior de 32dp.

Para lograr esto, abrimos el archivo fragment\_welcome.xml y creamos un LinearLayout de la siguiente manera:

<?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:layout\_margin="20dp"

android:orientation="vertical"

android:gravity="center\_horizontal" >

<TextView

android:id="@+id/welcome\_text"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:gravity="center\_horizontal"

android:textSize="20sp"

android:layout\_margin="20dp"

android:text="@string/welcome\_text" />

<Button

android:id="@+id/button\_start"

android:layout\_width="wrap\_content"

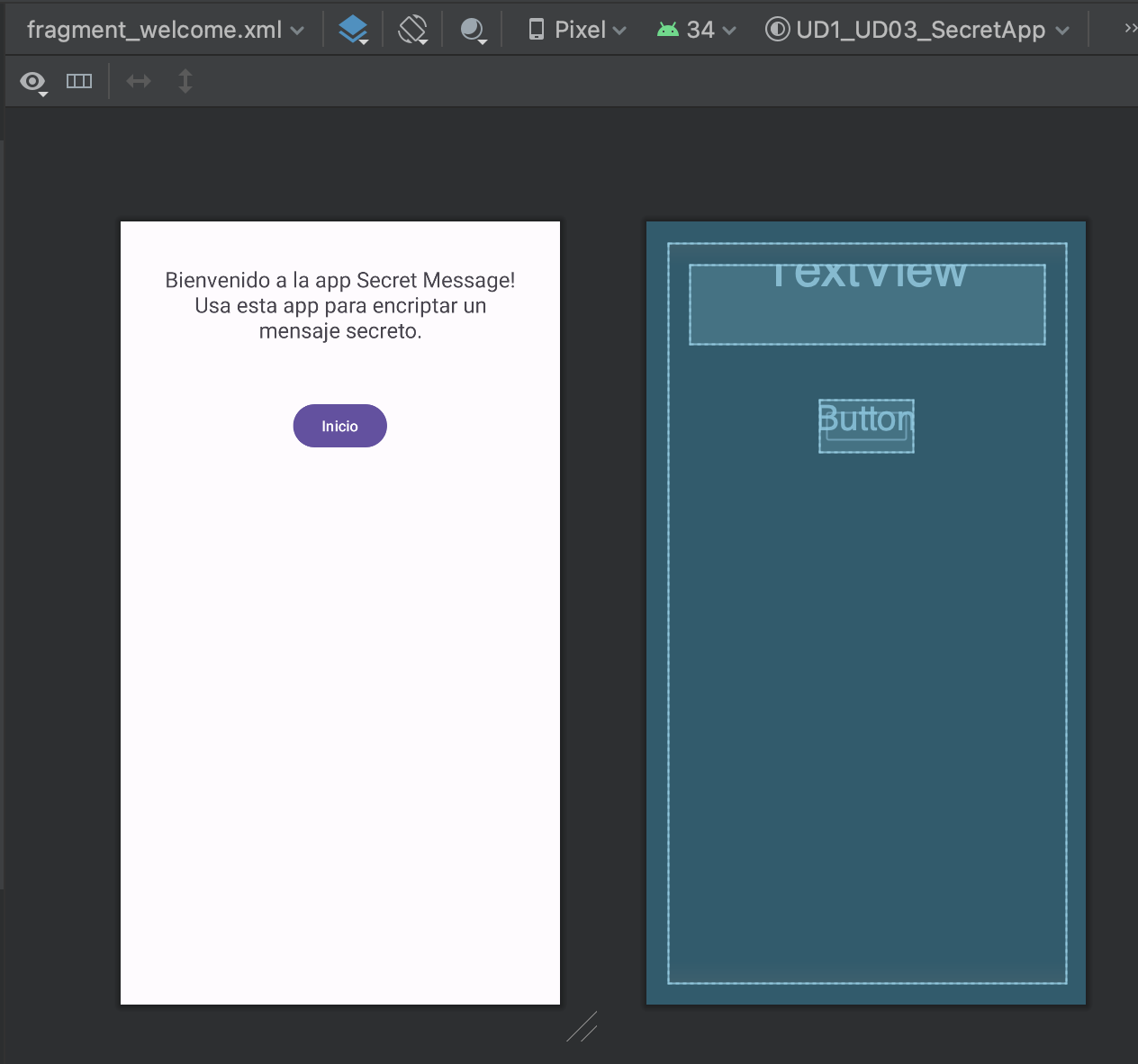
android:layout\_marginTop="32dp"

android:layout\_height="wrap\_content"

android:text="@string/start" />

</LinearLayout>

El resultado final se verá así:



# Fragment Container View

El fragmento que acabamos de crear se mostrará dentro de la actividad principal, que, a su vez, cuenta con su propio diseño (layout). En el archivo activity\_mail.xml, donde se define el diseño de la actividad principal, crearemos un contenedor de fragmentos llamado [FragmentContainerLayout](https://developer.android.com/reference/androidx/fragment/app/FragmentContainerView" \t "_blank). Este contenedor nos permite cargar fragmentos dinámicamente.

Para agregar esta funcionalidad, debemos modificar el archivo build.gradle y añadir la siguiente línea de dependencia:

implementation 'androidx.fragment:fragment-ktx:1.6.1'

En nuestro caso, dado que solo tendremos un fragmento que ocupará toda la pantalla, podemos utilizar directamente un FragmentContainerView. Sin embargo, debemos tener en cuenta que no podemos usarlo como elemento raíz de nuestro diseño. Para configurar los elementos básicos del contenedor y asignarle un identificador para futuras referencias, podemos definirlo de la siguiente manera en el archivo XML:

<?xml version="1.0" encoding="utf-8"?>

<androidx.fragment.app.FragmentContainerView

xmlns:tools="http://schemas.android.com/tools"

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

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:padding="16dp"

android:id="@+id/fragment\_container\_view">

A través de la propiedad “name”, podemos especificar qué fragmento se cargará en este contenedor. De esta manera, al modificar esta propiedad, podemos cambiar dinámicamente el fragmento que se muestra en función de las acciones realizadas en nuestra aplicación:

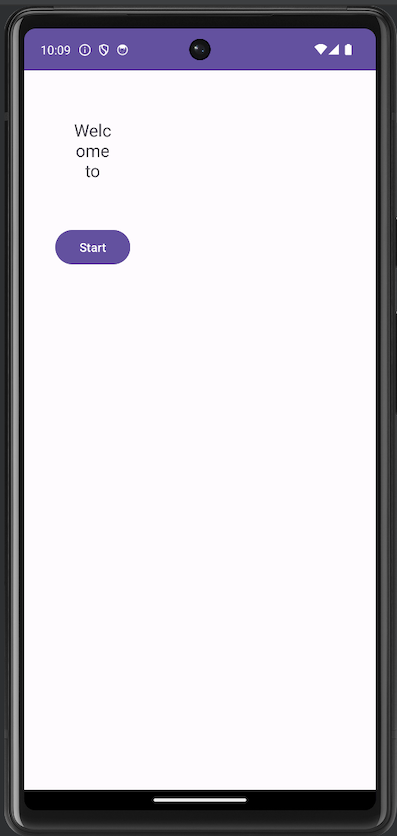
android:name="com.example.ud1\_ud03\_secretapp.WelcomeFragment"

# Flujo de ejecución

Actualmente, contamos con una actividad principal en nuestro archivo MainActivity.kt. Cuando esta actividad se crea, su principal tarea es construir la interfaz de usuario definida en el archivo XML asociado a activity\_main.xml.

En este momento, nuestro diseño se basa en un **“contenedor de fragmentos”**. Inicialmente, este contenedor de fragmentos carga un fragmento específico. Este fragmento opera de manera similar a una actividad, en el sentido de que tiene su propio archivo Kotlin con código que se ejecuta al lanzarse. Además, el fragmento crea su propia interfaz de usuario a partir de un archivo XML, siguiendo el mismo proceso que una actividad.

Sin embargo, al ejecutar la aplicación, nos encontramos con un problema visual, que se muestra en la siguiente imagen:

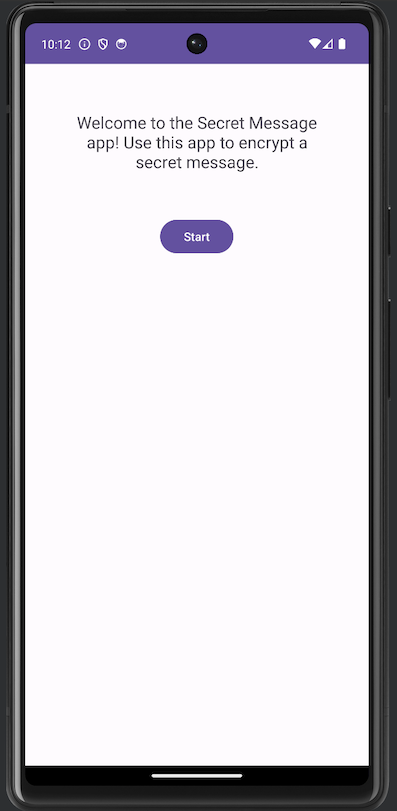


Este problema se debe a que el ancho y el alto del contenedor están configurados para ajustarse automáticamente al contenido (en este caso, el fragmento). Para resolverlo, debemos ajustar el ancho y el alto del contenedor para que coincidan con las dimensiones de la pantalla:

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

Al ejecutar la aplicación nuevamente, podemos comprobar que la interfaz se muestra correctamente:



 Idioma

Comprobar la configuración del idioma. Cambiar idioma del móvil al inglés para comprobar que funciona.

# Múltiples Fragmentos

Es posible incluir **más de un fragmento** en una misma pantalla. Para esto, vamos a crear un LinearLayout que contendrá dos contenedores de fragmentos:

<?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"

>

<androidx.fragment.app.FragmentContainerView

xmlns:tools="http://schemas.android.com/tools"

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:padding="16dp"

android:id="@+id/fragment\_container\_view"

android:name="com.example.ud1\_ud03\_secretapp.WelcomeFragment">

</androidx.fragment.app.FragmentContainerView>

<androidx.fragment.app.FragmentContainerView

xmlns:tools="http://schemas.android.com/tools"

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:padding="16dp"

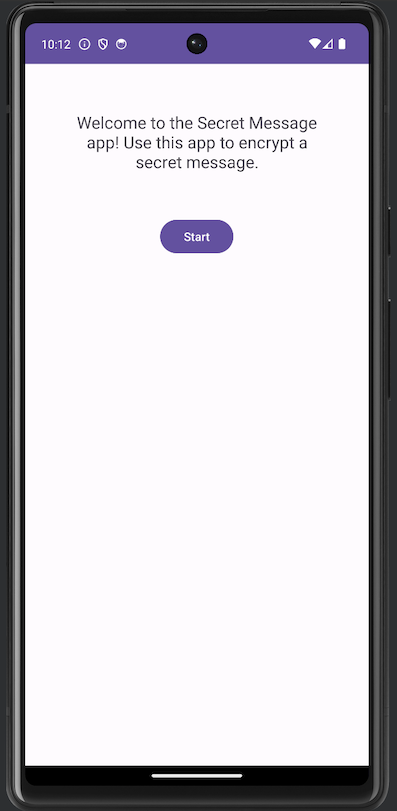
android:id="@+id/fragment\_container\_view\_2"

android:name="com.example.ud1\_ud03\_secretapp.WelcomeFragment">

</androidx.fragment.app.FragmentContainerView>

</LinearLayout>

Al ejecutar este diseño, observamos que solo se muestra un fragmento:



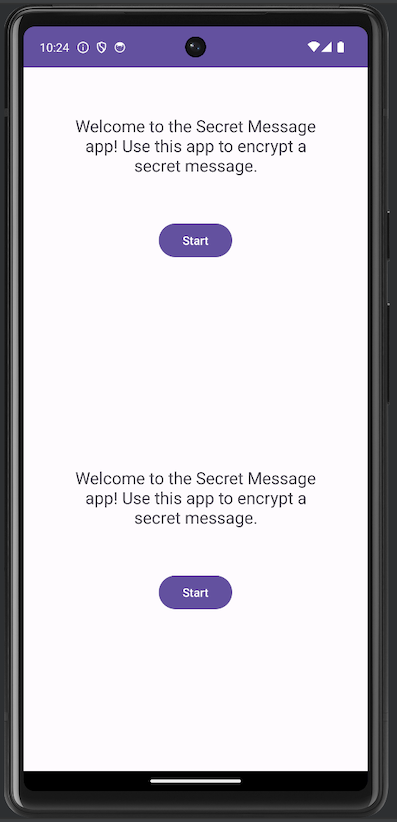
La razón de esto es que las propiedades de ancho (layout\_width) y alto (layout\_height) no están configuradas correctamente, ya que ambos fragmentos están tratando de ocupar todo el espacio del contenedor padre, que es el mismo para ambos. Para solucionarlo, simplemente debemos añadir la propiedad “layout\_weight” para que el espacio se divida de manera equitativa entre ambos fragmentos, asignándoles un 50% a cada uno:

<androidx.fragment.app.FragmentContainerView>

android:layout\_weight="1"

</androidx.fragment.app.FragmentContainerView>

Con esta configuración, ahora mostramos ambos fragmentos en la misma pantalla:



Es importante entender que cada uno de estos fragmentos se comporta como una **actividad independiente**, con su propio ciclo de vida y gestión de eventos, similar a tener múltiples pantallas que pueden funcionar de manera independiente en la misma interfaz.

 Alerta

Revertir los cambios de este apartado para tener un único contenedor.

Fragment Message

Crea un nuevo fragmento con la apariencia y características deseadas, sigue estas especificaciones:

**Layout:**

* Debe estar centrado horizontalmente en la pantalla.

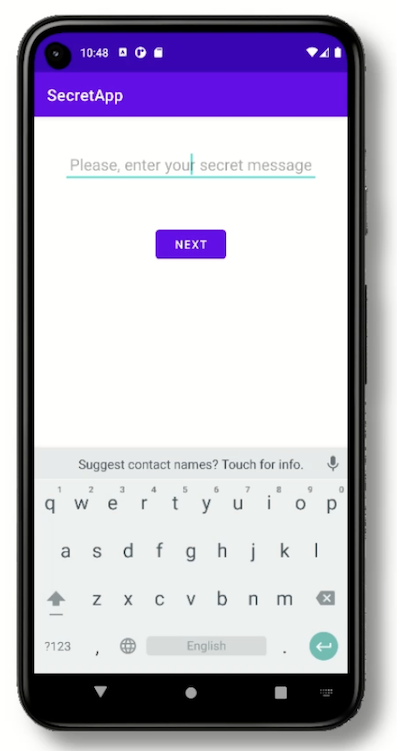
**EditText:**

* Debe ser de varias líneas.
* Debe tener un margen de 20dp.
* El tamaño del texto debe ser de 20sp.
* Debe mostrar un hint.

**Button:**

* Debe tener un margen superior (top) de 32dp.

El resultado final debe coincidir con la siguiente apariencia:



 Solución

De forma similar que en el caso anterior, creamos de nuevo un fragmento con el nombre MessageFragment. En este fragmento añadimos una caja de texto que nos permita escribir varias líneas y con un botón que nos indique “Siguiente”, como el diseño es muy sencillo podemos utilizar un LinearLayout:

<?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:layout\_margin="20dp"

android:orientation="vertical"

android:gravity="center\_horizontal" >

<EditText

android:id="@+id/edit\_text"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:textSize="20sp"

android:layout\_margin="20dp"

android:hint="@string/mensage\_text"

android:inputType="textMultiLine" />

<Button

android:id="@+id/button\_next"

android:layout\_width="wrap\_content"

android:layout\_marginTop="32dp"

android:layout\_height="wrap\_content"

android:text="@string/next" />

</LinearLayout>

Ahora vamos a editar el archivo Kotlin asociado al fragmento y nos quedamos únicamente con el método onCreateView.

class MessageFragment : Fragment() {

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

// Inflar el diseño (layout) para este fragmento

return inflater.inflate(R.layout.fragment\_message, container, false)

}

}

Para verificar que este fragmento funciona correctamente, vamos a realizar cambios en el archivo activity\_main.xml. Reemplazaremos el fragmento anterior con el nuevo fragmento que hemos creado:

<androidx.fragment.app.FragmentContainerView

xmlns:tools="http://schemas.android.com/tools"

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:padding="16dp"

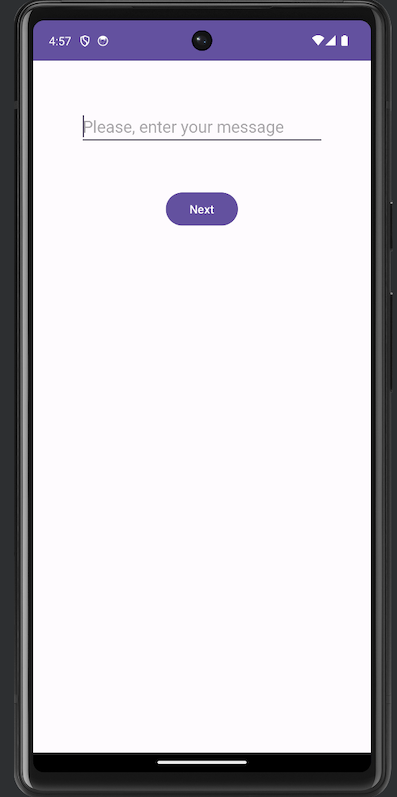
android:layout\_weight="1"

android:id="@+id/fragment\_container\_view"

android:name="com.example.ud1\_ud03\_secretapp.MessageFragment">

</androidx.fragment.app.FragmentContainerView>

Esto generará la siguiente apariencia en la pantalla:

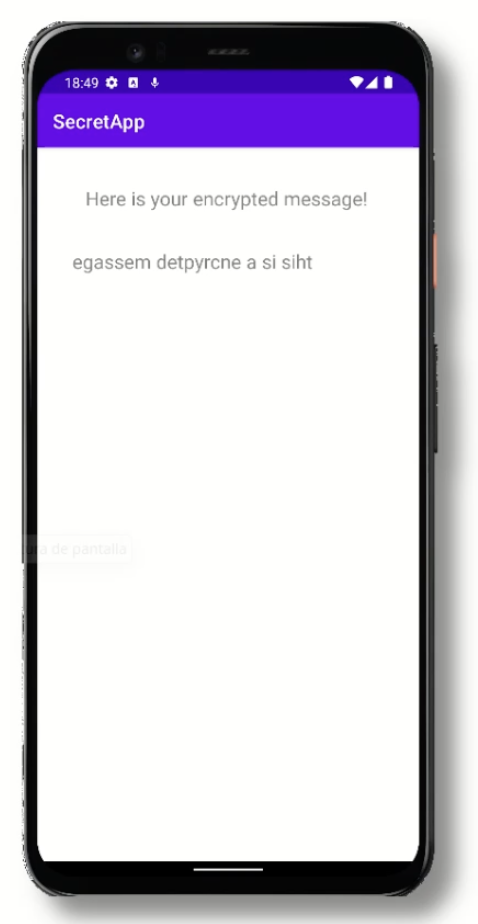


 Alerta

Para volver a cargar el fragmento “WelcomeFragment”, debes revertir los cambios realizados en esta sección del código en el archivo.

# Fragment Encrypt

Vamos a diseñar el tercer fragmento, que constituirá la pantalla donde se muestra el mensaje cifrado. En este caso, no tendremos un botón, ya que haremos uso del botón de Android para volver atrás y permitir que el usuario regrese a la pantalla inicial. Queremos lograr la siguiente apariencia:



Para construir esta pantalla, utilizaremos un LinearLayout con las siguientes especificaciones:

* Centrado horizontal.
* TextView con un margen de 20dp, centrado horizontal y un tamaño de texto de 20sp.
* Otro TextView con un margen de 20dp, un tamaño de texto de 20sp y capacidad de mostrar múltiples líneas.

Para lograr esto, primero debemos crear un nuevo fragmento en blanco al que llamaremos “EncryptFragment”. Luego, abrimos el archivo fragment\_encrypt.xml y creamos un LinearLayout con las siguientes propiedades:

<?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:layout\_margin="20dp"

android:orientation="vertical"

android:gravity="center\_horizontal">

<TextView

android:id="@+id/encrypt\_text"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:gravity="center\_horizontal"

android:textSize="20sp"

android:layout\_margin="20dp"

android:text="@string/encrypt\_text" />

<TextView

android:id="@+id/encrypt\_textView"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

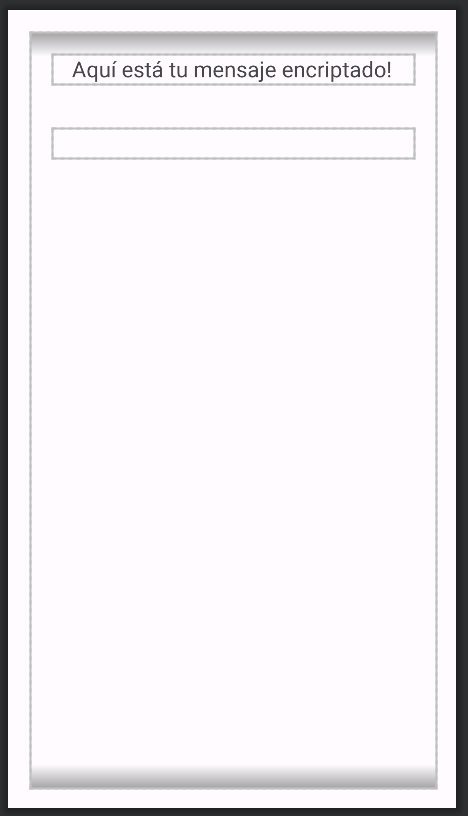
android:gravity="center\_horizontal"

android:textSize="20sp"

android:layout\_margin="20dp" />

</LinearLayout>

El resultado final se verá de la siguiente manera:



Luego, debemos modificar la lógica del fragmento y quedarnos únicamente con el método onCreateView:

class EncryptFragment : Fragment() {

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

// Inflamos el diseño de este fragmento

return inflater.inflate(R.layout.fragment\_encrypt, container, false)

}

}

Capitulo 2

# Navegación

Para la navegación entre fragmentos, utilizaremos el componente **Navigation** de Jetpack, conocido como el controlador de navegación, el cual administra qué fragmento se muestra en el host de navegación a medida que el usuario se desplaza por la aplicación.

La navegación entre fragmentos implica tres elementos clave:

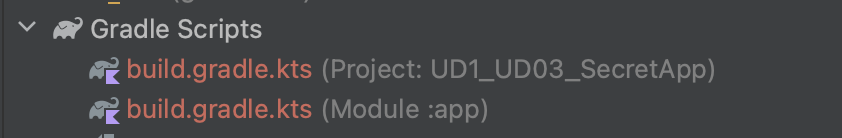
* **Grafo de navegación**: Este grafo describe las posibles “rutas” que el usuario puede seguir al navegar por la aplicación. Aunque se define como un recurso XML, por lo general se edita en el editor de diseño visual.
* **Host de navegación**: Este componente actúa como el contenedor en el que se presenta el fragmento al que estamos navegando. Se encuentra dentro del diseño de la actividad.
* **Controlador de navegación**: El controlador de navegación se encarga de determinar qué fragmento se mostrará en el host de navegación a medida que el usuario interactúa con la aplicación. Este controlador se manipula desde el código de la aplicación.

#### **Referencias**

* [Inicio con Navigation](https://developer.android.com/guide/navigation/navigation-getting-started?hl=es-419)
* [Navigation](https://developer.android.com/guide/navigation)

# Dependencia

Dado que este componente es una adición adicional, necesitamos incorporarlo a la configuración de nuestra aplicación. Para hacerlo, vamos a realizar modificaciones en el archivo de configuración build.gradle de nuestra aplicación y añadir la nueva dependencia. Aquí hay dos archivos disponibles, uno relacionado con la aplicación y otro con el proyecto en general. Elegiremos el archivo asociado a la aplicación, como se muestra a continuación:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud03/2.navegacion/1.dependencia/index.html#R-image-41fe5efbac666c185b700b432c726c86)

Luego, incluiremos la siguiente línea en la sección de dependencias:

dependencies {

// ... Otras dependencias ...

// Kotlin

implementation("androidx.navigation:navigation-fragment-ktx:2.7.7")

// ... Otras dependencias ...

}

Una vez que hayamos agregado esta línea, deberemos hacer clic en el botón “Sync Now” para actualizar las dependencias. De esta forma, tendremos las siguientes dependencias:

dependencies {

implementation("androidx.core:core-ktx:1.9.0")

implementation("androidx.appcompat:appcompat:1.6.1")

implementation("com.google.android.material:material:1.9.0")

implementation("androidx.navigation:navigation-fragment-ktx:2.7.7")

implementation("androidx.constraintlayout:constraintlayout:2.1.4")

testImplementation("junit:junit:4.13.2")

androidTestImplementation("androidx.test.ext:junit:1.1.5")

androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")

# Grafo de navegación

Como hemos visto, el grafo de navegación es un recurso XML que podemos agregar de manera sencilla. Para hacerlo, simplemente creamos un nuevo “Archivo de Recurso de Android” desde la carpeta de Recursos y seleccionamos el tipo “Navigation”.

GIF 21

Una vez creado, se abrirá automáticamente un editor visual que nos permitirá añadir nuestro grafo. Al hacer clic en el botón para agregar un nuevo destino, veremos una lista de los diferentes fragmentos que hemos creado.

GIF 22

Al seleccionar el botón para agregar otro fragmento, veremos que se añaden ambos fragmentos al editor visual.

GIF 23

Si hacemos clic en cualquiera de estos fragmentos, notaremos que se muestra un círculo que nos permite definir acciones, es decir, rutas que nos llevarán de un fragmento a otro.

GIF 24

Al seleccionar una de esas flechas, podemos ver sus atributos y confirmar que se trata de una acción que tiene un ID por defecto que utilizaremos más adelante.

Como mencionamos previamente, estos grafos son archivos XML, y podemos ver su código en la pestaña “Code”:

<?xml version="1.0" encoding="utf-8"?>

<navigation xmlns:android="http://schemas.android.com/apk/res/android"

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

android:id="@+id/nav\_graph"

app:startDestination="@id/welcomeFragment">

<fragment

android:id="@+id/welcomeFragment"

android:name="com.example.ud1\_ud03\_secretapp.WelcomeFragment"

android:label="WelcomeFragment">

<action

android:id="@+id/action\_welcomeFragment\_to\_messageFragment"

app:destination="@id/messageFragment" />

</fragment>

<fragment

android:id="@+id/messageFragment"

android:name="com.example.ud1\_ud03\_secretapp.MessageFragment"

android:label="MessageFragment" />

</navigation>

En el código anterior, es relevante destacar la propiedad app:startDestination="@id/welcomeFragment", que indica cuál es el primer fragmento que se carga. También podemos observar las definiciones de ambos fragmentos, así como la acción que conecta el primer fragmento con el segundo.

# NavHostFragment

Vamos a aprender cómo agregar el ***container*** que contendrá nuestro sistema de navegación. Si nos dirigimos al diseño de la actividad principal, encontramos un contenedor de fragmentos, donde ahora hemos añadido el fragmento “WelcomeFragment”.

Lo que necesitamos hacer es insertar un “host de navegación” en ese contenedor, es decir, un contenedor para nuestro sistema de navegación que, a su vez, contendrá los distintos fragmentos. Afortunadamente, la biblioteca de navegación de Android nos proporciona un componente de este tipo llamado NavHostFragment, que es un fragmento en sí mismo y, además, implementa toda la lógica relacionada con la navegación. Por lo tanto, podemos incluir directamente este fragmento de navegación en nuestro contenedor.

Para lograrlo, simplemente debemos realizar algunos cambios:

1. Primero, vamos a reemplazar el **fragmento** que tenemos actualmente:

android:name="com.example.ud1\_ud03\_secretapp.MessageFragment"

con este componente:

android:name="androidx.navigation.fragment.NavHostFragment"

NavHostFragment será nuestro “host de navegación” que contendrá los fragmentos por los que navegaremos. Como se puede observar en la [documentación oficial de este componente](https://developer.android.com/reference/androidx/navigation/fragment/NavHostFragment), NavHostFragment hereda de Fragment.

open class NavHostFragment : Fragment, NavHost

por eso lo podemos meter dentro de nuestro contenedor. Además, impmeneta una interface (NavHost) donde está la lógica relativa que nos permite desplazarnos entre fragmentos.

1. Debemos proporcionar el **grafo de navegación**. Para hacerlo, agregaremos un nuevo espacio de nombres (namespace), específico para la biblioteca de navegación:

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

Aprovechando este espacio de nombres, incluiremos dos nuevas propiedades:

app:navGraph="@navigation/nav\_graph"

app:defaultNavHost="true"

La primera propiedad hace referencia al grafo de navegación que creamos. La segunda propiedad, aunque no la utilizaremos por el momento, está relacionada con el botón “Atrás” en las aplicaciones; con esta opción podemos controlar la navegación a través de él.

Antes de continuar, verifiquemos que, tras los cambios realizados, la aplicación se carga correctamente y muestra el primer fragmento.

Navegación

El siguiente paso es habilitar la transición a la otra pantalla al presionar el botón “Start”. Para lograrlo, vamos a realizar cambios en el código del WelcomeFragment para obtener una referencia al botón y agregar un listener que defina el comportamiento de navegación hacia el otro fragmento.

Notamos que en este caso, el fragmento no hereda de la clase Activity, por lo que no podemos usar el método findViewByID directamente. En su lugar, obtendremos una referencia al botón de otra manera. Como este método onCreateView crea una **vista**, utilizaremos esa vista para encontrar el botón utilizando findViewByID. En lugar de devolver la vista de inmediato, la almacenaremos en una variable:

val view = inflater.inflate(R.layout.fragment\_welcome, container, false)

Ahora, dentro de esta variable view, tenemos acceso al método findViewByID, lo que nos permite obtener una referencia al botón “Start”:

val buttonStart = view.findViewById<Button>(R.id.button\_start)

Luego, podemos agregar un listener al botón y activar la acción definida en nuestro grafo de navegación. En el grafo, tenemos una acción que se ve así:

<action

android:id="@+id/action\_welcomeFragment\_to\_messageFragment"

app:destination="@id/messageFragment" />

Para lograr esta transición, utilizaremos el **controlador de navegación**, que es el tercer elemento que habíamos definido al principio:

* Grafo de navegación: representa el mapa de navegación.
* Host de navegación: integra el sistema de navegación.
* Controlador de navegación: a través de este, podemos activar las acciones.

Para obtener una referencia al controlador de navegación, lo hacemos a través de la vista utilizando el método findNavController(). Dentro de este controlador, tenemos un método llamado navigate, al cual simplemente le pasamos la acción que deseamos que se ejecute:

buttonStart.setOnClickListener {

view.findNavController().navigate(R.id.action\_welcomeFragment\_to\_messageFragment)

}

Si ejecutamos nuestra aplicación, verificamos que al presionar el botón “Start”, cambiamos de pantalla.

# Grado de Navegación Completo

Vams a vovler al grafo de navegación para completarlo. En este momento tenemos dos fragmentos: WelcomeFragment y MessageFragment. Ahora, vamos a añadir nuestro tercer fragmento y creamos una acción que va desde la segunda pantalla a este fragmento:

GIF 25

Recuerda que esta acción tiene un identificador que nos permitirá hacer referencia a ella:



Con estos pasos, habrás añadido el tercer fragmento a tu aplicación y establecido la navegación desde la segunda pantalla a la pantalla de cifrado.

A continuación, deberemos agregar el **listener** asociado al botón para poder cambiar de pantalla. Para hacerlo, nos dirigimos al Fragmento de Mensajes (MessageFragment) y, de manera similar a lo que hicimos anteriormente, creamos una referencia al botón y definimos la acción a ejecutar:

val view = inflater.inflate(R.layout.fragment\_message, container, false)

val buttonNext = view.findViewById<Button>(R.id.button\_next)

buttonNext.setOnClickListener {

view.findNavController().navigate(R.id.action\_messageFragment\_to\_encryptFragment)

}

return view

Si ejecutamos nuestra aplicación en este punto, veremos que la navegación de la aplicación funciona de manera completa.

GIF 26

Capitulo 3

# Argumentos

Para intercambiar información entre diferentes fragmentos, utilizaremos un plugin de Gradle llamado [Safe Args](https://developer.android.com/guide/navigation/use-graph/safe-args?hl=es-419" \t "_blank), el cual se integra con el componente de navegación. Este componente asegura la integridad de los tipos, por lo tanto, se recomienda su uso de manera general. El funcionamiento de este componente es bastante simple.

En primer lugar, debemos crear los **argumentos** en el **fragmento** que recibirá la información. Este componente generará dos clases en todos los fragmentos que tengan acciones definidas, es decir, fragmentos que se dirijan hacia cualquier otro destino dentro de nuestra aplicación.

Se generará una clase llamada “**Directions**”. El nombre por defecto de esta clase será el nombre del propio fragmento, seguido de “Directions”. En nuestro ejemplo, con el fragmento “Message”, se generará “MessageDirections”.

En cuanto a la recepción de datos en los fragmentos que tienen argumentos definidos y, por lo tanto, recibirán datos, se creará una clase llamada **“Args”** siguiendo la convención mencionada anteriormente, es decir, el nombre del fragmento seguido de “Args”. Esta clase tendrá un método llamado **“fromBundle()”** que básicamente nos permitirá definir un Bundle en el cual se almacenarán los datos que se pasen como argumento al fragmento. Podremos recuperar estos datos desde ese Bundle.

Este mecanismo es bastante **sencillo** y, gracias a la funcionalidad de Safe Args, se construirán automáticamente estas clases a través de las cuales podemos enviar la información y la clase desde la cual la recibiremos. Veremos esto en detalle en nuestro proyecto.

# Dependencia

Para utilizar este componente, debemos incorporar un plugin en nuestro proyecto, lo que implica realizar ajustes en dos archivos Gradle, tanto en el nivel del **proyecto** como en la **aplicación**.

En el archivo de configuración de nivel superior (el del proyecto), donde podemos agregar opciones de configuración comunes para todos los subproyectos/módulos, se deben incluir las siguientes líneas:

plugins {

alias(libs.plugins.android.application) apply false

alias(libs.plugins.jetbrains.kotlin.android) apply false

id("androidx.navigation.safeargs") version "2.5.3" apply false

}

Una vez que hayamos realizado estas modificaciones en el archivo, es necesario sincronizar las dependencias haciendo clic en el botón **“Sync Gradle”** o **“Sincronizar dependencias”,** según la interfaz de desarrollo que estemos utilizando.

Además, en el archivo de configuración Gradle de la **aplicación**, debemos agregar el plugin de la siguiente manera:

plugins {

alias(libs.plugins.android.application)

alias(libs.plugins.jetbrains.kotlin.android)

id("androidx.navigation.safeargs")

}

Una vez hecho esto, habremos añadido el plugin de Gradle a nuestro proyecto. Nuevamente, será necesario **sincronizar** las dependencias haciendo clic en el botón “Sync Gradle” para asegurarnos de que todas las configuraciones se apliquen correctamente.

# Definir Argumentos

Vamos a dirigirnos a nuestro grafo y, en el fragmento final, llamado **“EncryptFragment”**, que será el **destinatario final** de nuestro texto, vamos a incorporar un argumento. Llamaremos a este argumento “Message” y le asignaremos el tipo “String”.

GIF 27

En el proceso, cada fragmento que tenga una acción definida en **SafeArgs** generará una clase llamada “Directions” y creará un método para cada una de las acciones (flechas) que hayamos definido. Los parámetros de estos métodos serán la lista de argumentos que hemos creado. En este caso, solamente tenemos un argumento llamado “Message”.

# Pasar Argumentos

Ahora, vamos a cambiar el código en el fragmento **“MessageFragment”** para transmitir el mensaje al fragmento siguiente. En lugar de invocar directamente la siguiente pantalla al generar el Listener para el botón “Siguiente”, vamos a crear una **acción** a partir de la clase generada automáticamente llamada “MessageFragmentDirections”. Observa que esta clase tiene un método con el mismo nombre que la acción que hemos definido en el grafo, y como parámetros, tiene un argumento con el mismo nombre que el que definimos en el grafo.

// Crear una acción para la navegación

val actions = MessageFragmentDirections.actionMessageFragmentToEncryptFragment()

// Para obtener el texto ingresado por el usuario desde la caja de texto

val message = view.findViewById<EditText>(R.id.edit\_text).text.toString()

// Pasar el mensaje como argumento a la acción

val actions = MessageFragmentDirections.actionMessageFragmentToEncryptFragment(message)

// Finalmente, transmitir esta acción al controlador de navegación

view.findNavController().navigate(actions)

En realidad, podríamos simplificar aún más este proceso, pasando todo como argumento directamente:

view.findNavController().navigate(

MessageFragmentDirections.actionMessageFragmentToEncryptFragment(

view.findViewById<EditText>(R.id.edit\_text).text.toString()

)

)

Este enfoque más conciso facilita la navegación y asegura que el mensaje se transmita correctamente al fragmento de destino.

# Recuperar Argumentos

Para recuperar los valores transmitidos entre fragmentos, debemos dirigirnos al archivo Kotlin correspondiente, en este caso, “EncryptMessage.kt”. Antes que nada, definiremos una **variable** para almacenar la **vista** que se construirá en este fragmento, lo que nos permitirá acceder a sus diferentes **elementos**. Lo que nos interesa especialmente en este momento es acceder al campo de texto que contendrá el texto cifrado.

val view = inflater.inflate(R.layout.fragment\_encrypt, container, false)

return view

Para recuperar los elementos, aprovecharemos la clase “Args” que el plugin crea automáticamente. Esta clase ofrece varios métodos, pero el que nos interesa es **“fromBundle”**, el cual nos permitirá acceder al bundle donde se almacenan los datos. La clase “Fragment” también cuenta con un método llamado **“requireArguments”** que nos proporcionará un bundle con los diferentes **valores**.

A partir del bundle, tendremos automáticamente generados una serie de atributos que coincidirán con el argumento que definimos en el grafo. En este caso, para acceder al mensaje transmitido, podemos utilizar el siguiente código:

val mensajeCifrado = EncryptFragmentArgs.fromBundle(requireArguments()).message

Luego, podemos guardar este valor en una constante y posteriormente pasar este valor a una función que se encargue de cifrarlo y mostrarlo en la interfaz.

# Práctica Optativa. Creación de Mapa Camino de Santiago

Desarrolla una aplicación llamada “UD03\_3\_MiCamino” que integre un **mapa interactivo** utilizando la **API de Google Maps** en Android. La aplicación mostrará los puntos marcados a lo largo del Camino Francés de Santiago de Compostela. Los requisitos son los siguientes:

1. **Mapa Interactivo**: Al abrir la aplicación, se mostrará un mapa centrado en la ruta del Camino Francés, indicando las paradas existentes en el mismo.
2. **Optativo. Descripción de los Puntos**: Cada marcador debe tener una breve descripción que se muestre al hacer clic en él, así como la opción de obtener direcciones desde la ubicación actual del usuario hasta el punto de interés.
3. **Optativo. Marcadores de Puntos de Interés**: Implementa marcadores en el mapa que representen puntos clave a lo largo del Camino Francés, como albergues, iglesias, y otros lugares de interés. Optativo: Al hacer clic en un marcador, se mostrará un InfoWindow con información adicional sobre el lugar.

### **Requisitos adicionales:**

* Utiliza Kotlin.
* Implementa una interfaz de usuario intuitiva que facilite la navegación entre el mapa y la lista de puntos de interés.

### **Recursos:**

* Sigue el [codelab de Google](https://developers.google.com/codelabs/maps-platform/maps-platform-101-android" \l "0" \t "_blank) como guía para la implementación de la funcionalidad del mapa y los marcadores.

Práctica. Creación de Cuentos de Halloween Personalizados

**Descripción de la Práctica:**

En esta práctica, crearás una aplicación de Android UD03\_2\_Halloween que permitirá a los usuarios generar cuentos de Halloween personalizados. La aplicación constará de tres pantallas, cada una implementada como un fragmento.

**Pantalla 1: Introducción de Nombre**

* En esta pantalla, los usuarios podrán introducir su nombre en un cuadro de texto.
* Deberás incluir una etiqueta para indicar que deben escribir su nombre.
* Un botón de “Continuar” les llevará a la siguiente pantalla una vez que hayan ingresado su nombre.

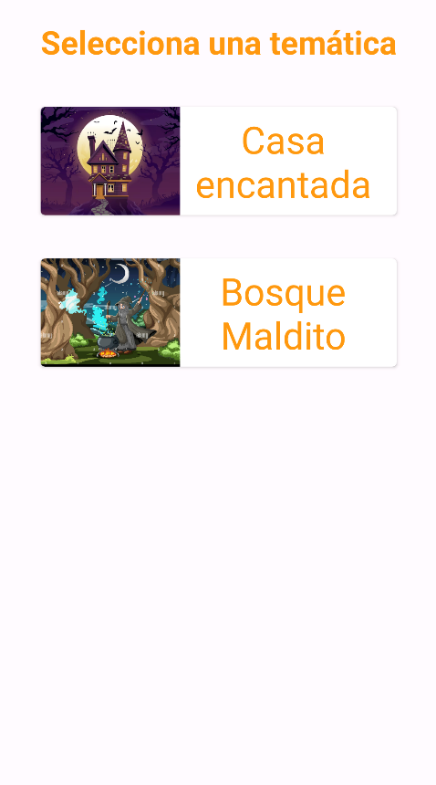
Tendrá el siguiente aspecto:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud03/6.practica/index.html#R-image-ece87d8044149c009f23cca9ce15207b)

**Pantalla 2: Selección de Temática**

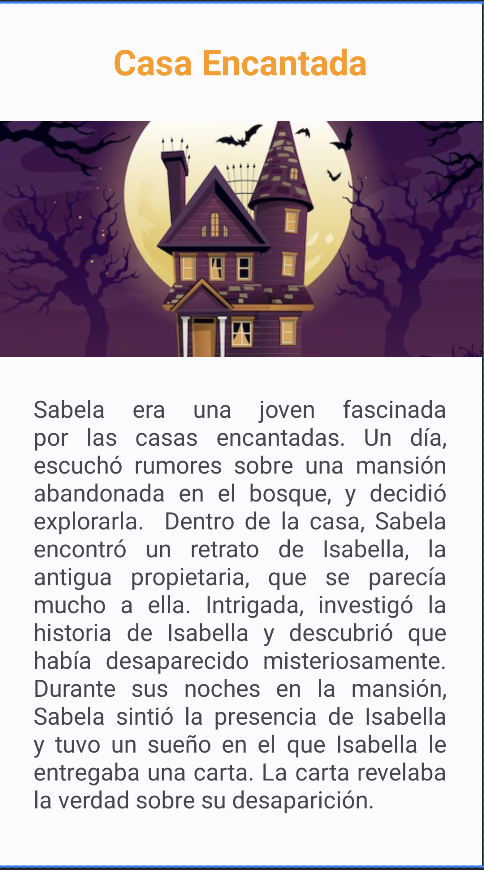
* En esta pantalla, los usuarios podrán elegir una temática para su cuento de Halloween. Debes proporcionar al menos tres opciones, como “Casa Encantada”, “Bosque Oscuro”, “Mansión Misteriosa”, entre otras.
* Cada opción debe presentarse como una CardView donde incluirás el texto y una imangen asociada.
* Al pulsar en alguna de las categorías, les llevará a la tercera pantalla después de hacer su elección.

Tendrá un aspecto similar a este:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud03/6.practica/index.html#R-image-3ba9cc2fe61b49682aca30d787e84a42)

**Pantalla 3: Cuento Personalizado**

* En esta pantalla, se generará un cuento personalizado basado en el nombre del usuario y la temática seleccionada.
* Deberás incluir una vista de texto para mostrar el cuento completo.
* Los detalles del cuento deben cambiar según el nombre ingresado y la temática seleccionada.
* Puedes agregar la imagen relacionada con la temática.
* Un botón de “Volver” en la parte inferior permitirá a los usuarios regresar a la pantalla de inicio para crear otro cuento si lo desean.

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud03/6.practica/index.html#R-image-2afe3ef264f7becf2b738f1b9a0e5406)

**Requisitos Técnicos:**

* Deberás utilizar fragmentos para implementar cada una de las pantallas.
* La interacción entre las pantallas debe gestionarse adecuadamente, de modo que la información ingresada por el usuario se utilice para generar el cuento personalizado.
* La aplicación debe tener un diseño atractivo y temático de Halloween.

**Consideraciones Adicionales:**

* Puedes utilizar recursos como imágenes o descripciones para cada temática de cuento.
* Asegúrate de que la aplicación sea fácil de usar y que los usuarios puedan seguir el flujo lógico de introducir su nombre, seleccionar una temática y leer su cuento personalizado.

Capitulo 4

# UD04. Navegación UI

### **Resultados de avaliación**

**RA1.** Aplica tecnoloxías de desenvolvemento para dispositivos móbiles, e avalía as súas características e as súas capacidades.

**RA2.** Desenvolve aplicacións para dispositivos móbiles, para o que analiza e emprega as tecnoloxías e as librarías específicas.

### **Criterios de avaliación**

* CA1.7 Analizouse a estrutura de aplicacións existentes para dispositivos móbiles, e identificáronse as clases utilizadas.
* CA1.8 Realizáronse modificacións sobre aplicacións existentes.
* CA1.9 Utilizáronse emuladores para comprobar o funcionamento das aplicacións.
* CA2.1 Xerouse a estrutura de clases necesaria para a aplicación.
* CA2.4 Utilizáronse as clases necesarias para a conexión e a comunicación con dispositivos sen fíos.
* CA2.5 Utilizáronse as clases necesarias para o intercambio de mensaxes de texto e multimedia.
* CA2.8 Realizáronse probas de interacción entre o usuario e a aplicación para mellorar as aplicacións desenvolvidas a partir de emuladores.
* CA2.9 Empaquetáronse e despregáronse as aplicacións desenvolvidas en dispositivos móbiles reais.
* CA2.10 Documentáronse os procesos necesarios para o desenvolvemento das aplicacións

### **BC1. Análise de tecnoloxías para desenvolvemento de aplicacións en dispositivos móbiles.**

* Estrutura dunha aplicación para dispositivo móbil.
* Modificación de aplicacións existentes.
* Uso do contorno de execución do administrador de aplicacións.
* Contornos integrados de traballo.
* Módulos para o desenvolvemento de aplicacións móbiles.
* Emuladores.
* Ferramentas e fases de construción.
* Eventos da interface.
* Probas de interacción.
* Empaquetaxe e distribución.
* Documentación do desenvolvemento das aplicacións.
* Comunicacións: clases asociadas. Tipos de conexións.

Crear proyecto

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
   * Selecciona “Start a new Android Studio project” en la pantalla de inicio.
   * Elige “Phone and Tablet” como tipo de dispositivo.
   * Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
   * En la siguiente pantalla, completa la información básica sobre el proyecto:
     + **Name**: Ingresa “UF1\_UD04\_1\_CatChat”.
     + **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
     + **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
     + **Language**: Selecciona “Kotlin” como lenguaje de programación.
     + **Minimum API level**: Selecciona “API 35: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
   * Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
   * Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad MainActivity.kt y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

Capitulo 1

# Navegación UI

Continuaremos profundizando en el componente de **navegación** y en el **desplazamiento** de los distintos fragmentos que conforman nuestra aplicación. En este punto, nos enfocaremos en ciertos elementos que nos permitirán crear interfaces de usuario más enriquecidas y proporcionar mecanismos más flexibles de desplazamiento.

Hasta ahora, los proyectos que hemos desarrollado tenían una **navegación lineal** en la que el desplazamiento entre ellos siempre seguía el mismo orden. Ahora exploraremos una serie de componentes que nos permitirán desplazarnos **libremente** entre estos distintos elementos que forman parte de nuestra aplicación. Específicamente, veremos cómo aplicar **temas** a nuestra aplicación, y en algunos de estos temas, cómo se definen el uso de determinadas barras de aplicación, como la que aparece en la parte superior de todos los proyectos que hemos estado creando hasta ahora, donde se muestra el título de la aplicación, por ejemplo. También aprenderemos a reemplazar estas barras por una barra de botones personalizada, que contendrá botones específicos para saltar a fragmentos particulares o para retroceder, entre otras funcionalidades.

Además, abordaremos la creación de **menús** que incluirán una serie de elementos que nos permitirán navegar hacia los diversos destinos que tengamos definidos en nuestra aplicación. Por último, exploraremos la implementación de una barra de navegación inferior, un elemento común en las aplicaciones de Android.

# Nuevo Diseño

Vamos a crear una nueva aplicación con la siguiente apariencia:



Al iniciar un nuevo proyecto, automáticamente se genera una **barra de navegación** en la parte **superior** que muestra el **título** de la aplicación. Si ejecutamos la aplicación tal como está, notaremos que esta barra superior ya incluye el título de la aplicación y tiene un color específico.

Este comportamiento se debe al hecho de que, al crear nuestro proyecto, se aplica un tema específico que define el diseño de la aplicación. En este caso, hemos utilizado un tema basado en **Material Design de Google**, que asegura un diseño coherente. El objetivo es que todas las aplicaciones que desarrollemos para el sistema Android tengan una **apariencia similar**, lo que facilita la experiencia del usuario al encontrar interfaces familiares. Puedes obtener más información sobre el diseño Material Design [aquí](https://m3.material.io/get-started).

# Temas y Estilos

En nuestro archivo Gradle de la aplicación, podemos observar que se ha incluido una dependencia:

implementation("com.google.android.material:material:1.10.0")

Esta dependencia indica que estamos incorporando este componente en nuestra aplicación de manera predeterminada.

Además, en el archivo de **manifiesto** de la aplicación, hemos especificado el tema o estilo que nuestra aplicación utilizará, definiendo aspectos como el **color** primario y secundario, entre otros:

android:theme="@style/Theme.UF1\_UD04\_1\_CatChat"

El tema “UF1\_UD04\_1\_CatChat” está definido en un archivo de recursos ubicado en la carpeta res -> values -> themes. En esta ubicación, encontramos **dos temas** que se aplicarán en función de la configuración de **día** o **noche**. Este tema hereda de un tema predefinido en Material Design:

<resources xmlns:tools="http://schemas.android.com/tools">

<!-- Base application theme. -->

<style name="Base.Theme.UF1\_UD04\_1\_CatChat" parent="Theme.Material3.DayNight.NoActionBar">

<!-- Personaliza tu tema claro aquí. -->

<!-- <item name="colorPrimary">@color/my\_light\_primary</item> -->

</style>

<style name="Theme.UF1\_UD04\_1\_CatChat" parent="Base.Theme.UF1\_UD04\_1\_CatChat" />

</resources>

En este punto, es importante destacar que no se ha definido una barra de acción (NoActionBar). Sin embargo, si modificamos esta línea para que herede de la clase “DarkActionBar”:

<style name="Base.Theme.UF1\_UD04\_1\_CatChat" parent="Theme.MaterialComponents.DayNight.DarkActionBar">

y ejecutamos nuestra aplicación, notaremos que ahora se incluye una barra de navegación en la parte superior de la pantalla.

Es posible realizar ajustes en ciertos parámetros, como colores y estilos, que se pueden encontrar en la documentación oficial de Android. No obstante, generalmente se recomienda no modificar estos parámetros en exceso para mantener la coherencia en el estilo de las aplicaciones.

Por ejemplo:

<item name="colorPrimary">@color/my\_light\_primary</item>

En un tema que admite modos día y noche, es importante aplicar las modificaciones en ambos archivos de tema correspondientes a cada modo.

# Toolbar Personalizada

Ahora procederemos a reemplazar la barra por defecto, heredando del tema base “NoActionBar” (que es el tema predeterminado).

Para garantizar que ambos archivos de temas heredan de “NoActionBar”, asegurémonos de que ambos tienen la siguiente definición:

<style name="Base.Theme.UF1\_UD04\_1\_CatChat" parent="Theme.Material3.DayNight.NoActionBar">

Por el momento, no realizaremos modificaciones en los colores que vienen por defecto. No obstante, si en el futuro deseáramos modificar algún color específico, podemos definirlos en el archivo `colors.xml``.

### **Componente Toolbar**

Vamos a agregar un nuevo componente a nuestro archivo principal (activity\_main.xml), específicamente, vamos a incorporar un componente “Toolbar”. Para lograr esto, podemos utilizar un simple “LinearLayout”. En primer lugar, eliminamos el contenido existente en activity\_main.xml y creamos un “LinearLayout” vacío:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

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

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical"

>

</LinearLayout>

Dentro de este “LinearLayout”, vamos a introducir un nuevo componente “**MaterialToolbar**” y ajustaremos su ancho al tamaño de la pantalla, y su altura la configuraremos para que coincida con el valor predeterminado de la barra del tema actual. Para lograr esto, utilizaremos el atributo de altura con el valor “?attr/actionBarSize”:

<com.google.android.material.appbar.MaterialToolbar

android:layout\_width="match\_parent"

android:layout\_height="?attr/actionBarSize"

android:id="@+id/toolbar"

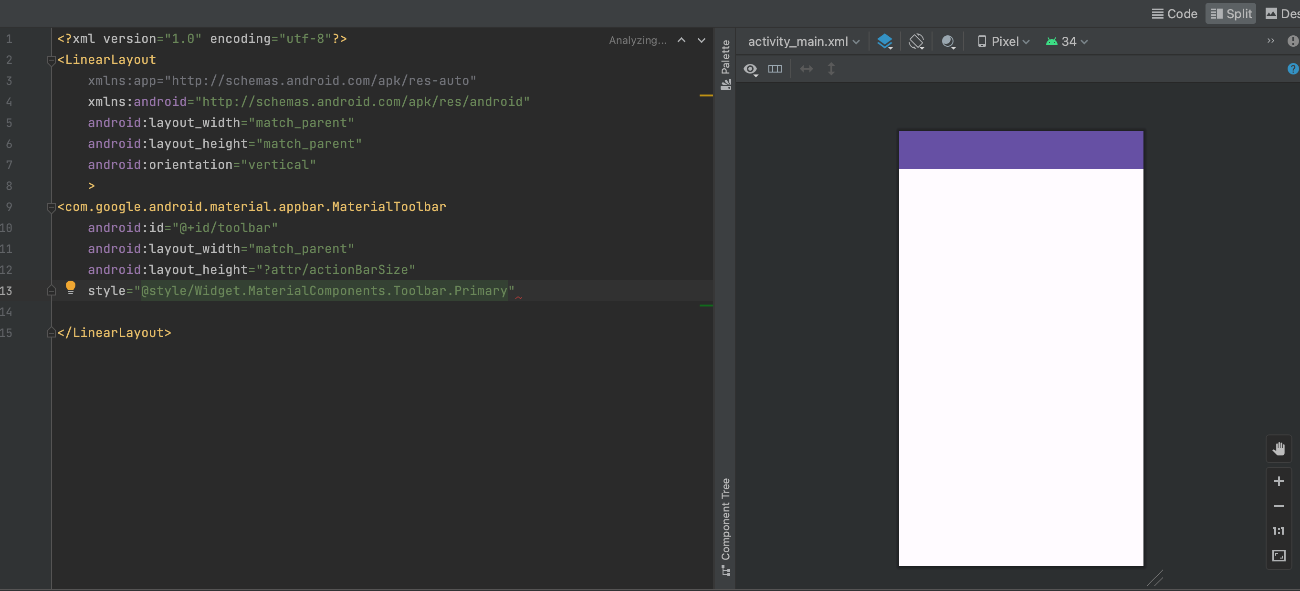
style="@style/Widget.MaterialComponents.Toolbar.Primary"

/>

Además, es necesario asignarle un identificador (en este caso, “toolbar”) para que podamos hacer referencia a él desde nuestro código.

Con el atributo “style,” aplicamos el estilo definido en el tema por defecto, accediendo a él mediante la referencia @style/Widget.MaterialComponents.Toolbar.Primary.

Al visualizar la pestaña de diseño, ya podremos ver la barra de navegación con el color de la barra primaria:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/1.navegacion_ui/3.toolbar_personalizada/index.html#R-image-dfd00240cb22b6e81e717c4e65d32442)

Es importante mencionar que, en este momento, la barra todavía carece de contenido, ya que acabamos de crearla dentro del diseño.

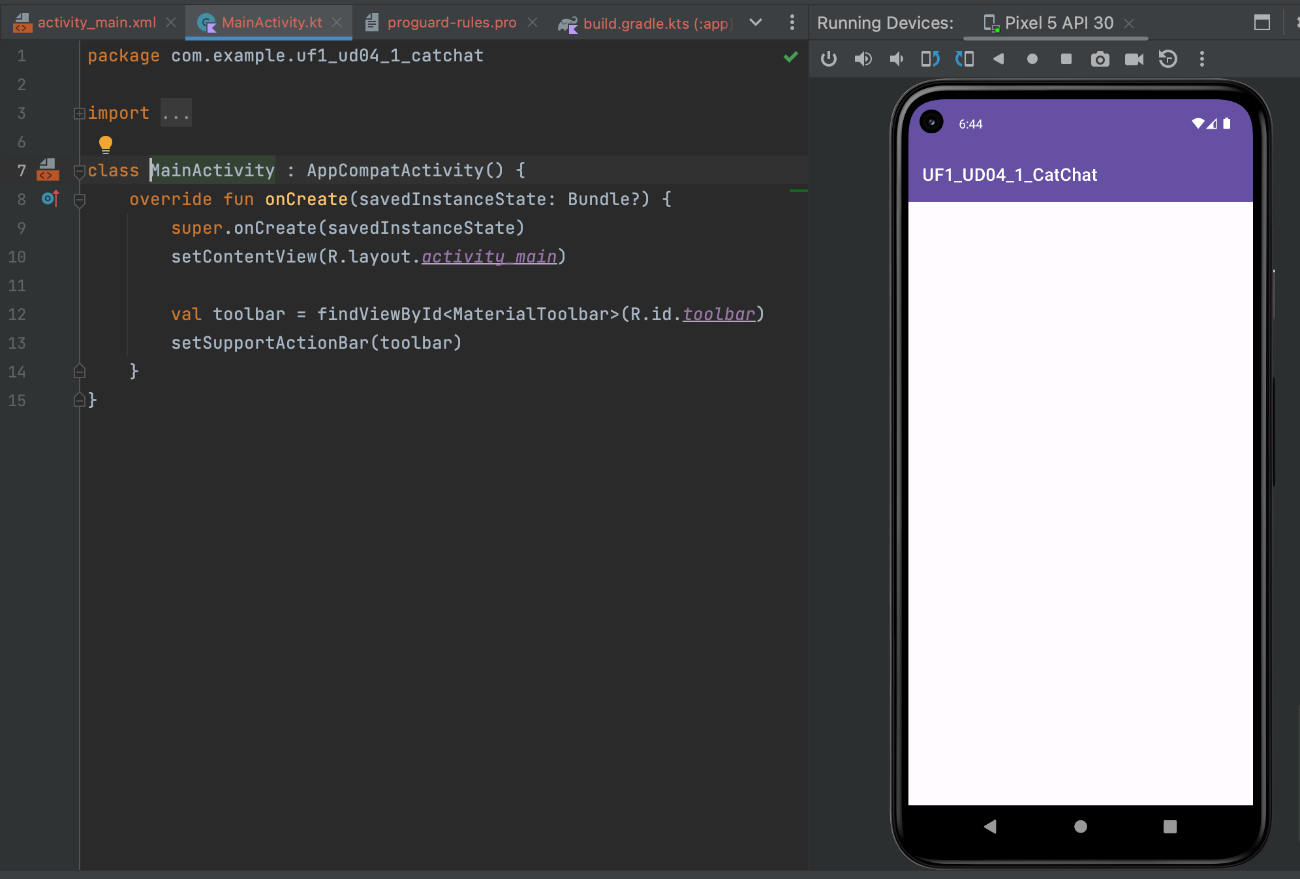
### **Asignar Activity**

En este punto, necesitamos asignar la **barra de navegación** que hemos creado como la barra de **aplicación principal**. Para lograr esto, en nuestra actividad principal (ActivityMain.kt), debemos recuperar la instancia de la barra que acabamos de crear y especificar que la utilice como la barra de aplicación principal:

val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)

setSupportActionBar(toolbar)

Una vez completado este paso, si ejecutamos la aplicación, veremos que la barra de navegación se ha personalizado y que ahora podemos agregar nuevos elementos a ella:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/1.navegacion_ui/3.toolbar_personalizada/index.html#R-image-22daead351ef0c02fbb4ec61d60569e1)

Esta personalización nos permite integrar nuevos elementos en la barra y adaptarla a las necesidades de nuestra aplicación.

# Fragmentos

Antes de poder añadir elementos a la barra de navegación, vamos a crear los fragmentos que vamos a utilizar. Ambos fragmentos son muy simples y consisten en una etiqueta que indica en qué pantalla nos encontramos.

El resultado final debe coincidir con la siguiente apariencia:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/1.navegacion_ui/4.fragmentos/index.html#R-image-e71faf1dbde28491d89bf62978c71d13)

#### **Fragmentos Inbox**

Primero, creamos un fragmento llamado “InboxFragment.” En este fragmento, agregamos un simple diseño utilizando un LinearLayout:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<TextView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:text="Inbox"

android:textSize="20dp"

/>

</LinearLayout>

#### **Fragmentos Help**

De manera similar, creamos otro fragmento llamado “HelpFragment.” Al igual que en el caso anterior, utilizamos un LinearLayout para un diseño simple:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<TextView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:text="Help"

android:textSize="20dp"

/>

</LinearLayout>

#### **Fragmento Sent Items**

De manera similar, creamos otro fragmento llamado “MessagesFragment.” Al igual que en el caso anterior, utilizamos un LinearLayout para un diseño simple:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<TextView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:text="Sent Items"

android:textSize="20dp"

/>

</LinearLayout>

Una vez completados los fragmentos, editamos el archivo Kotlin asociado al fragmento y mantenemos solo el método onCreateView:

class MessageFragment : Fragment() {

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

// Inflar el diseño (layout) para este fragmento

return inflater.inflate(R.layout.fragment\_message, container, false)

}

}

Estos fragmentos simples servirán como las pantallas de “Inbox”, “Help” y “Message” en nuestra aplicación.

# Grafo de Navegación

Además de los fragmentos, necesitamos crear un nuevo grafo de navegación que incluya las pantallas que estamos desarrollando, siguiendo el mismo enfoque que vimos en la unidad anterior.

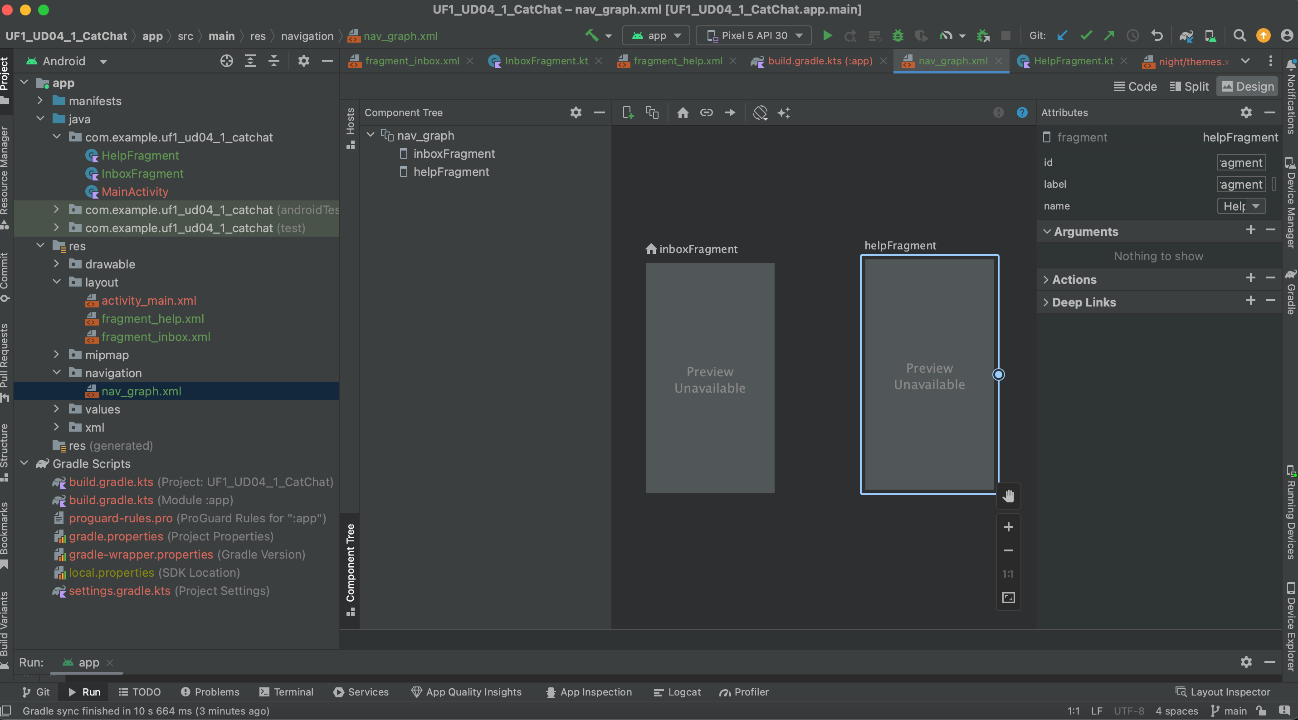
Para comenzar, debemos agregar la dependencia de navegación en el archivo Gradle:

implementation("androidx.navigation:navigation-fragment-ktx:2.7.4")

Después de agregar esta línea, es importante hacer clic en el botón “Sync Now” para actualizar las dependencias de Gradle.

Una vez que tengamos la dependencia añadida, podemos proceder a crear el grafo de navegación. Como hemos visto, el grafo de navegación es un recurso XML que se crea de manera sencilla. Para ello, creamos un nuevo **“Archivo de Recurso de Android”** desde la carpeta de Recursos, seleccionamos el tipo “Navigation,” y le damos el nombre “nav\_graph.xml.”

Dentro de este archivo, agregamos los dos fragmentos que hemos creado, pero por ahora no necesitamos crear acciones para la navegación entre ellos. El grafo debe verse de la siguiente manera:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/1.navegacion_ui/5.grafo_navegacion/index.html#R-image-15c98378857fdf6021455c9fc43222ab)

En el código, notamos que a cada fragmento se le ha asignado un identificador:

<fragment

android:id="@+id/inboxFragment"

android:name="com.example.uf1\_ud04\_1\_catchat.InboxFragment"

android:label="Inbox" />

<fragment

android:id="@+id/helpFragment"

android:name="com.example.uf1\_ud04\_1\_catchat.HelpFragment"

android:label="Help" />

Después de definir los destinos en el grafo, es importante que modifiquemos el diseño de la actividad principal (main\_activity.xml) para agregar un contenedor de fragmentos, ajustando su alto y ancho al tamaño de la pantalla y asignándole un identificador:

<androidx.fragment.app.FragmentContainerView

android:id="@+id/nav\_host\_fragment"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

En este caso, no tendremos un único fragmento en nuestro contenedor de pantalla, sino **varios fragmentos** definidos en el grafo de navegación. Para lograr esto, debemos definir un **“host de navegación”** en el contenedor de fragmentos, especificando el grafo que acabamos de crear:

android:name="androidx.navigation.fragment.NavHostFragment"

app:navGraph="@navigation/nav\_graph"

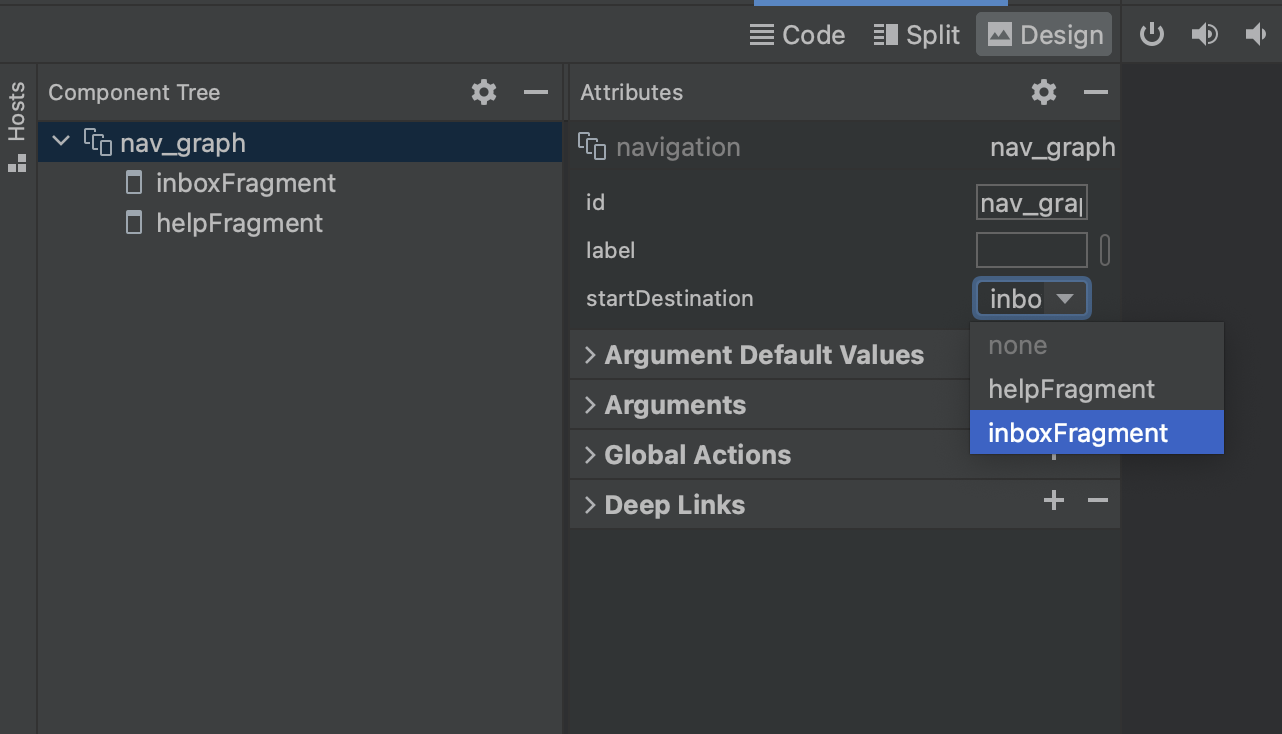
De esta forma, indicamos que este contenedor contendrá fragmentos que están definidos en el grafo de navegación. Además, como hicimos en el proyecto anterior, debemos establecer la propiedad para que el botón “atrás” funcione correctamente:

app:defaultNavHost="true"

Finalmente, podemos ejecutar la aplicación para verificar que se carga correctamente:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/1.navegacion_ui/5.grafo_navegacion/index.html#R-image-cb4208b2f96b8a373341ee62a536bf4a)

Comprobamos que la pantalla de “Inbox” se carga, ya que así lo hemos definido en el grafo de navegación:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/1.navegacion_ui/5.grafo_navegacion/index.html#R-image-48e0bb77f8f33f349a70afe0419ce992)

Este conjunto de pasos nos permite configurar la navegación en nuestra aplicación.

Capitulo 2

Toolbar

Para agregar elementos en la barra de navegación que permitan navegar entre los distintos fragmentos de la aplicación, necesitamos crear un recurso de tipo **menú**, que es otro tipo de recurso que se puede definir en Android.

Dentro de este menú, agregaremos diversos **elementos**, en este caso, botones que nos permitirán cargar los diferentes fragmentos. Es importante que el ID de cada **ítem** coincida con el ID del **fragmento de destino**, el cual se encuentra definido en el **grafo de navegación**.

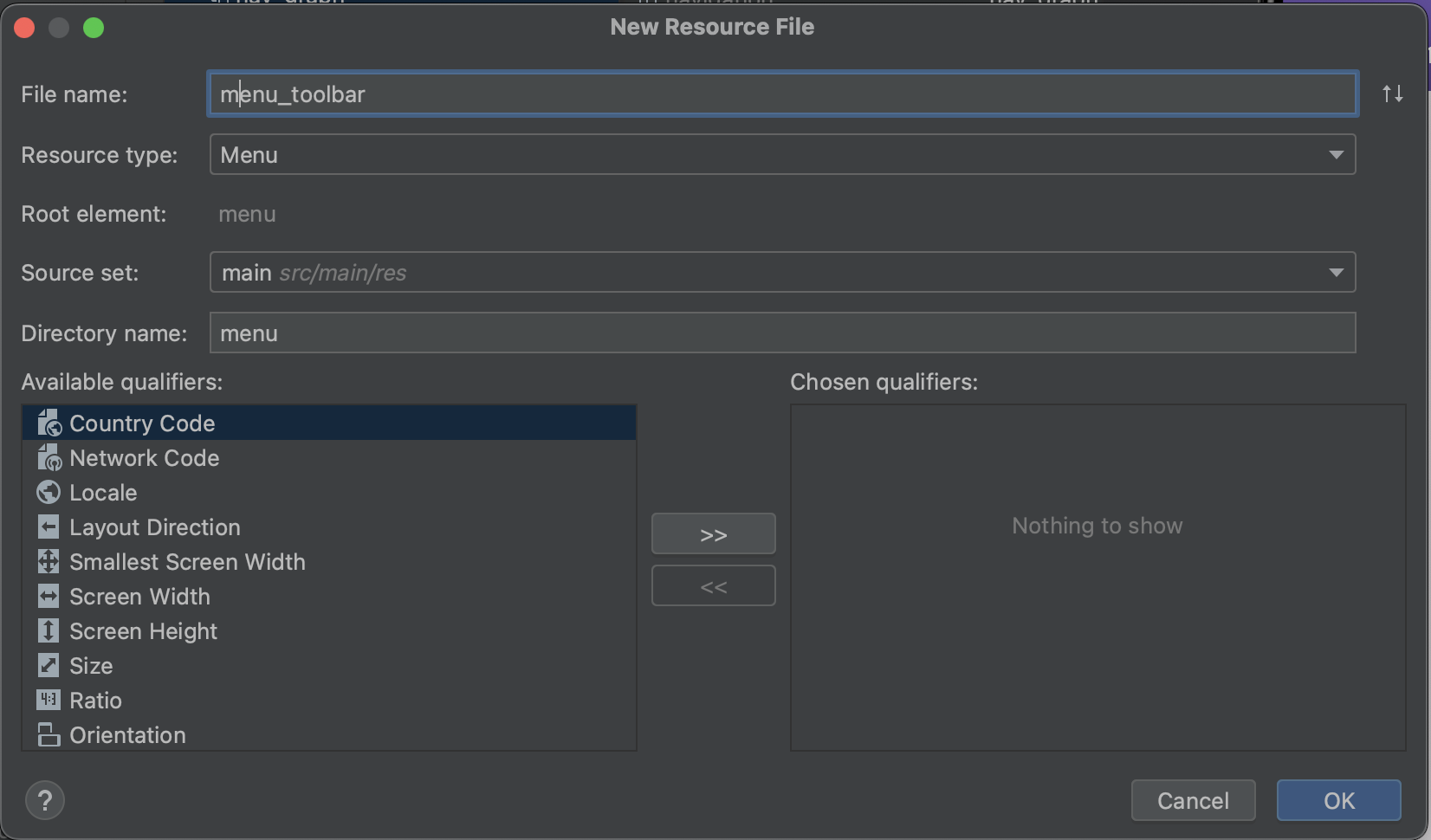
Una vez que tengamos este menú creado, para añadir los elementos del menú a la barra de navegación, debemos sobrescribir el método onCreateOptionsMenu en la actividad principal.

Luego, para responder a los eventos de clic en los elementos del menú, sobrescribiremos el método onOptionsItemSelected en la actividad principal. Este método nos permite manejar las acciones que ocurren cuando se hace clic en los elementos del menú.

Por último, para mostrar el label del fragmento (definido en el grafo de navegación) y un botón que nos permita retroceder, podemos utilizar la clase AppBarConfiguration proporcionada por la biblioteca de navegación, que es una forma sencilla de lograrlo.

# Creación del menú

Lo primero que haremos es crear el menú de opciones que vamos a asignar al toolbar. Para hacer esto, creamos un nuevo recurso llamado “menu\_toolbar” de tipo **“Android Resource File”** de tipo **Menú**, tal como se muestra en la imagen:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/2.toolbar/1.creacion_menu/index.html#R-image-33cad915b4dd157ed56ec28f6526e687)

Dentro de este menú, podemos definir diferentes opciones, denominadaos “**items**”. Comenzaremos por agregar la primera opción, “Help”. Para ello, agregamos un nuevo elemento con su **identificador**. Es importante que este ID sea el mismo que el del fragmento al que queremos navegar, ya que al hacer clic en este ícono, queremos saltar a la pantalla “HelpFragment,” por lo que seleccionamos el mismo ID que está definido en el grafo de navegación:

<?xml version="1.0" encoding="utf-8">

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

<item

android:id="@+id/helpFragment"

Otra configuración que deseamos aplicar es el **icono**. Existen varios íconos predefinidos para diferentes usos dentro de la propiedad “drawable,” y seleccionamos el ícono de ayuda:

android:icon="@android:drawable/ic\_menu\_help"

Además, proporcionamos un texto descriptivo, un **título**:

android:title="Help"

Por defecto, el menú se configura como un menú **desplegable**, lo cual no es lo que deseamos. Podemos modificar este comportamiento utilizando la propiedad “**showAsAction**” del espacio de nombres de la app. Primero, definimos el espacio de nombres de la app:

<?xml version="1.0" encoding="utf-8"?>

<menu

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

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

Luego, dentro del elemento del ítem, configuramos “showAsAction” como “always” para mostrarlo como un botón en lugar de una lista desplegable de opciones de menú:

<item

android:id="@+id/helpFragment"

android:icon="@android:drawable/ic\_menu\_help"

android:title="Help"

app:showAsAction="always" />

Aunque en el diseño del menú aparezca configurado correctamente, al ejecutar la aplicación, no vemos el botón generado en ningún lugar, como se muestra en la imagen:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/2.toolbar/1.creacion_menu/index.html#R-image-1c0de9107725d027835bfbe8e321d969)

Esto sucede porque la actividad principal (MainActivity) no está creando estos elementos del menú. Para resolver esto, debemos ir a la actividad principal y utilizar los métodos que mencionamos al principio.

# Inflado de Items

Dentro de la actividad principal (MainActivity.kt), realizaremos el inflado de los elementos del menú utilizando la función onCreateOptionsMenu:

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)

setSupportActionBar(toolbar)

}

override fun onCreateOptionsMenu(menu: Menu?): Boolean {

menuInflater.inflate(R.menu.menu\_toolbar, menu)

return super.onCreateOptionsMenu(menu)

}

}

De manera similar a cómo inflamos el diseño XML en la actividad principal, utilizamos la clase menuInflater con el método inflate para inflar el diseño del menú. Le pasamos el diseño de nuestro menú y el propio menú que recibe como parámetro para que lo “pinte” en la barra de navegación.

Al ejecutar la aplicación, veremos que ahora muestra el botón que diseñamos:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/2.toolbar/2.inflado_items/index.html#R-image-24d494c160ac207834d5def173277635)

Aunque por ahora, el botón no realiza ninguna acción.

**Prueba:** Puedes eliminar la propiedad app:showAsAction="always", y comprobarás cómo se muestra de manera diferente, posiblemente como un ítem desplegable en lugar de un botón.

# Acciones

Para responder a las acciones de clic sobre los botones del menú, debemos sobrescribir el método onOptionsItemSelected, donde obtendremos información sobre el elemento del menú que se presionó.

Primero, necesitamos obtener una referencia al controlador de navegación. Para ello, utilizamos la siguiente función:

val navController = findNavController(R.id.nav\_host\_fragment)

Para vincular el botón sobre el cual se hizo clic con el evento del controlador de navegación, debemos hacer uso de una nueva función, pero primero debemos agregar una nueva dependencia:

implementation("androidx.navigation:navigation-ui-ktx:2.7.4")

Después de agregar esta línea, hacemos clic en el botón “Sync Now” para actualizar las dependencias.

Ahora, en lugar de simplemente devolver el itemSelected, realizamos una acción utilizando el componente NavigationUI:

return NavigationUI.onNavDestinationSelected(item, navController) || super.onOptionsItemSelected(item)

Este método, onNavDestinationSelected, toma dos parámetros: el elemento del menú que se accionó y el controlador de navegación. También devuelve un valor booleano, que utilizamos en una operación lógica “or” y devolvemos al método padre.

La llamada a onNavDestinationSelected establece una relación entre el botón seleccionado y el host de navegación correspondiente.

El método onOptionsItemSelected debería verse así:

override fun onOptionsItemSelected(item: MenuItem): Boolean {

val navController = findNavController(R.id.nav\_host\_fragment)

return NavigationUI.onNavDestinationSelected(item, navController) || super.onOptionsItemSelected(item)

}

Ahora, si ejecutamos la aplicación, comprobaremos que al hacer clic en el botón de ayuda, nos llevará a la pantalla correcta.

Capitulo 3

Botton Navigation Bar

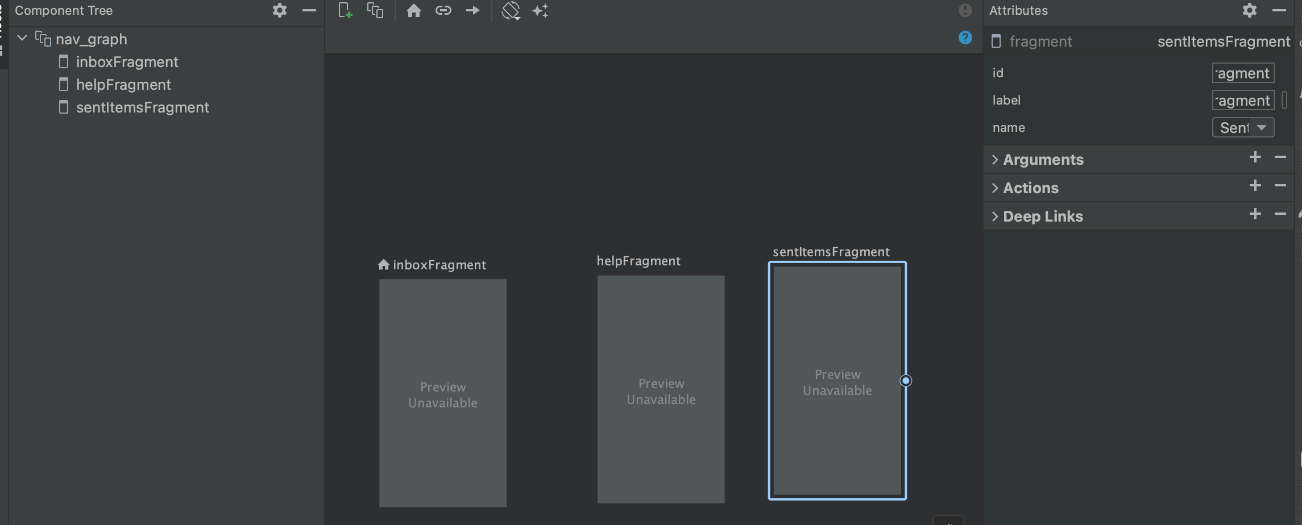
La barra de **navegación inferior**, conocida como Bottom Navigation Bar, es un componente de interfaz de usuario que se coloca en la parte inferior de la pantalla.

Para agregarla al diseño de la aplicación, se utiliza la vista BottomNavigationView, a la cual se le asigna el recurso de menú correspondiente mediante la propiedad “menu”.

Es importante tener en cuenta que el BottomNavigationView tiene una **limitación**, por la cual es que solo puede contener un máximo de 5 elementos de navegación.

# Grafo de Navegación

A continuación, debemos agregar este fragmento al grafo de navegación como se muestra en la imagen:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/3.bottonbar/1.grafo_navegacion/index.html#R-image-e6fcf10eda17bbcd0cbbba126576282e)

Además, modificaremos la etiqueta para que muestre un texto específico:

<fragment

android:id="@+id/sentItemsFragment"

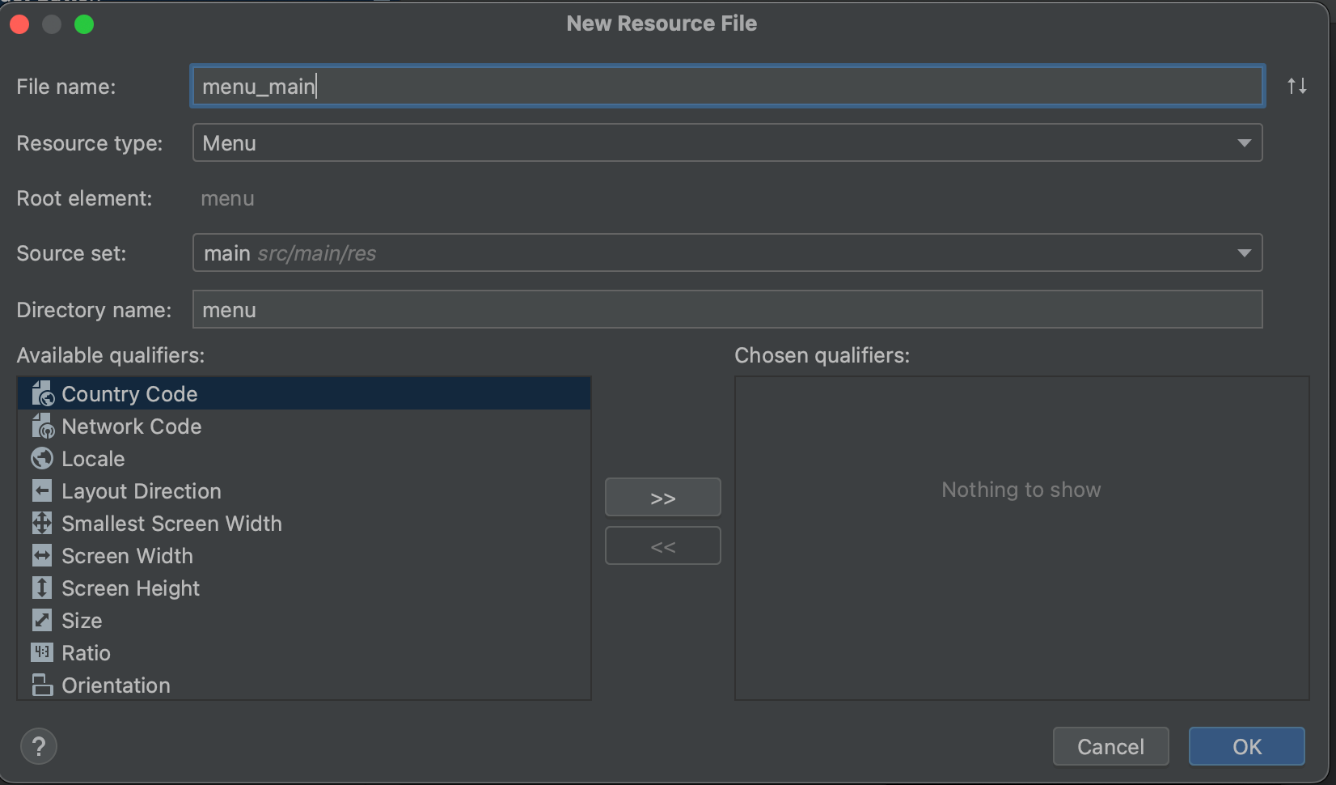
android:name="com.example.uf1\_ud04\_1\_catchat.SentItemsFragment"

android:label="Sent Items" />

Esto agregará la capacidad de navegar a la pantalla “Sent Items” en nuestra aplicación.

# Menú

Vamos a crear el menú inferior. Para ello, necesitamos agregar un nuevo archivo de recurso de tipo menú con el nombre “menu\_main,” como se muestra en la imagen:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/3.bottonbar/2.menu/index.html#R-image-27f7253466d3ba318153006a1e0b3eed)

Este menú consta de tres elementos que corresponden a las tres opciones del menú que deseamos crear:

<?xml version="1.0" encoding="utf-8"?>

<menu

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

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

<item

android:id="@+id/inboxFragment"

android:icon="@android:drawable/ic\_dialog\_email"

android:title="Inbox"

app:showAsAction="always"></item>

<item

android:id="@+id/sentItemsFragment"

android:icon="@android:drawable/ic\_menu\_send"

android:title="Sent Items"

app:showAsAction="always"></item>

<item

android:id="@+id/helpFragment"

android:icon="@android:drawable/ic\_menu\_help"

android:title="Help"

app:showAsAction="always"></item>

</menu>

Esto generará tres opciones en el menú inferior:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/3.bottonbar/2.menu/index.html#R-image-ae29d9ddb27e91d1ee0c3aa6e67a6f9b)

# Botton Navigation Bar

Para agregar el componente de la barra de navegación **inferior** a nuestro diseño, debemos realizar los siguientes pasos en el archivo activity\_main.xml. Primero, agregamos un componente de tipo BottomNavigationView, estableciendo un ID, el ancho y el alto, y asignando el menú que acabamos de crear:

<com.google.android.material.bottomnavigation.BottomNavigationView

android:id="@+id/bottom\_navigation"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

app:menu="@menu/menu\_main" />

Sin embargo, al observar el diseño de nuestra aplicación, es posible que el menú inferior no se muestre, esto se debe a que el contenedor de fragmentos ocupa todo el espacio en el diseño. Para que se muestre el menú inferior, podemos asignar un peso de 1 al contenedor de fragmentos, lo que permite que el espacio se distribuya entre los componentes. Modificamos el **FragmentContainerView** de la siguiente manera:

<androidx.fragment.app.FragmentContainerView

android:id="@+id/nav\_host\_fragment"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:name="androidx.navigation.fragment.NavHostFragment"

app:navGraph="@navigation/nav\_graph"

android:layout\_weight="1"

app:defaultNavHost="true" />

Ahora podemos verificar que el menú inferior se muestra correctamente en el diseño.

Controlador de Navegación

Para habilitar la funcionalidad de los botones, debemos vincular la **barra de navegación inferior** (BottomNavigation) con el **sistema de navegación**. Para lograr esto, acudimos al archivo MainActivity.kt y necesitamos dos cosas:

1. Obtener una referencia al nuevo componente en el diseño.

val bottomNav = findViewById<BottomNavigationView>(R.id.bottom\_navigation)

1. Luego, utilizamos esta referencia para vincularla al controlador de navegación:

bottomNav.setupWithNavController(navController)

En este caso, el proceso es más sencillo que en el caso anterior. Si ejecutamos la aplicación en este punto, comprobamos que todo funciona correctamente:

Capitulo 4

Navigation Drawer

Si necesitamos incluir más de 5 elementos en nuestra barra de navegación inferior y superar esta limitación, una solución es sustituirla por un “navigation drawer,” que no tiene esta restricción. El “navigation drawer” es un panel **ocultable** deslizable con **capacidad** para más elementos de menú.

Para implementarlo, agregaremos un componente "DrawerLayout" a la raíz del diseño de nuestra actividad principal. Este “DrawerLayout” contendrá dos vistas principales:

1. Una vista (layout) que representa la pantalla principal, generalmente el “navigation host”.
2. Un “NavigationView” que contendrá el contenido del panel de navegación.

Cuando el panel de navegación está cerrado, el “DrawerLayout” se comporta como una actividad normal, a excepción de un ícono que permite abrir el panel.

Cuando el panel de navegación está abierto, se desliza sobre el contenido de la actividad principal, revelando el menú de acciones correspondiente.

Para personalizar el menú del “navigation drawer,” podemos realizar las siguientes acciones:

* Agregar elementos de **menú** con títulos y, si es necesario, separadores.
* Agrupar elementos utilizando el elemento “**group**”. Esto permite definir comportamientos comunes, como el resaltado del elemento seleccionado (atributo “android:checkableBehavior”).
* Asignar un recurso de tipo menú al **panel de navegación**.

Para construir el panel del “navigation drawer,” podemos definir una **cabecera** (header) que nos permita agregar una imagen u otro recurso visual. Esto se logra mediante un layout que se integra en el panel de navegación.

Finalmente, para que el “navigation drawer” sea completamente funcional, debemos realizar dos pasos importantes:

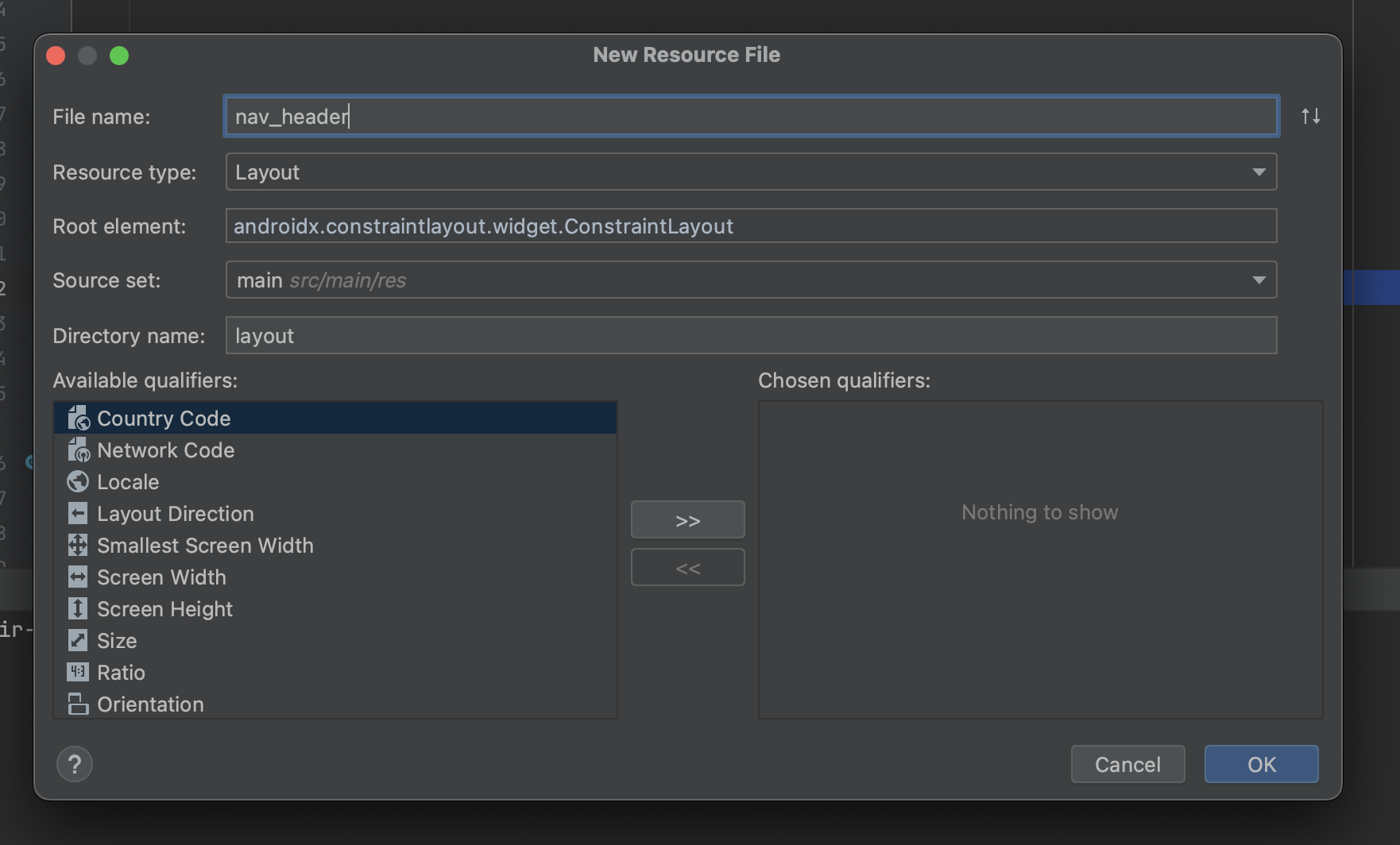
1. Agregarlo a la barra de herramientas (toolbar) de la aplicación para que el usuario pueda abrir y cerrar el panel.
2. Vincularlo con el controlador de navegación, de manera que al seleccionar un elemento del menú, la aplicación navegue a la pantalla correspondiente.

Esta implementación del “navigation drawer” brinda una mayor flexibilidad en la cantidad de elementos que podemos incluir en la barra de navegación y ofrece una experiencia de usuario más completa en aplicaciones con una amplia variedad de secciones o características.

# Header

Primero, crearemos el diseño de la **cabecera del menú**. Aunque esta cabecera es **opcional**, la utilizaremos en este ejemplo. Comenzaremos por descargar la [imagen](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/4.navigation_drawer/1.header/gato.webp) que se utilizará en la cabecera y la añadiremos a la carpeta “drawable” de nuestro proyecto.

Luego, crearemos un nuevo archivo de recursos de Android de tipo Layout con el nombre “nav\_header”:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/4.navigation_drawer/1.header/index.html#R-image-acc8a349bda9cf30e9a30bb4c17517c2)

Al crear el archivo, se generará un diseño basado en ConstraintLayout. Sin embargo, para este caso, queremos un diseño más sencillo, por lo que cambiaremos el ConstraintLayout a un FrameLayout. Ajustaremos el ancho para que ocupe todo el contenedor, pero estableceremos una altura fija de 180dp para que no ocupe toda la pantalla. El diseño contendrá una ImageView para mostrar la imagen que hemos añadido previamente.

<?xml version="1.0" encoding="utf-8"?>

<FrameLayout

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

android:layout\_width="match\_parent"

android:layout\_height="180dp">

<ImageView

android:layout\_width="wrap\_content"

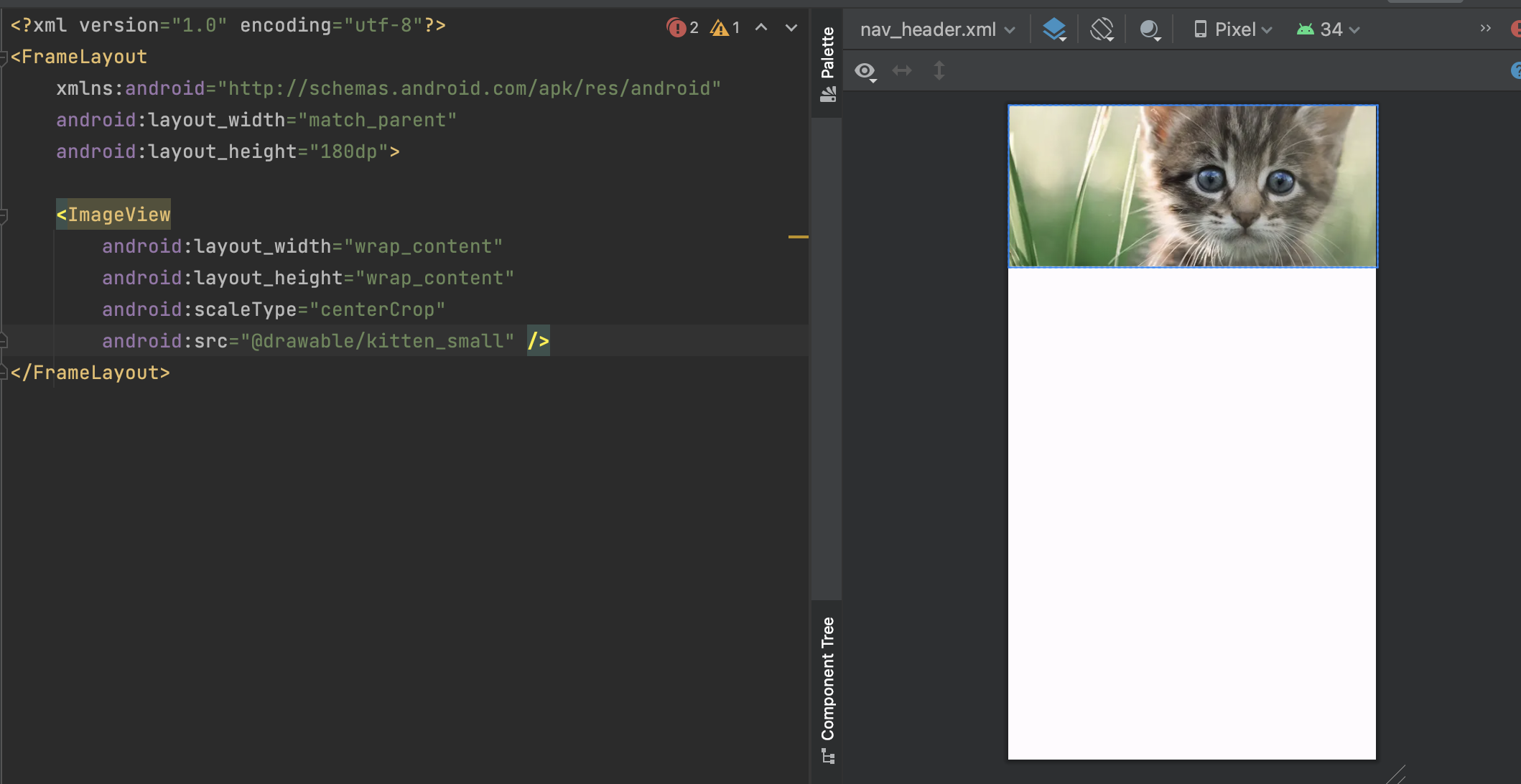
android:layout\_height="wrap\_content"

android:scaleType="centerCrop"

android:src="@drawable/kitten\_small" />

</FrameLayout>

Con este código, obtendremos el diseño de la cabecera con la imagen que hemos añadido.

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/4.navigation_drawer/1.header/index.html#R-image-765a4d1751670fca31a12004087f3e67)

Este diseño se usará como la **cabecera** de nuestro menú de navegación. Puedes personalizarlo más según tus preferencias o necesidades, como añadir texto o estilos adicionales.

# Menú

Vamos a aprovechar el diseño del **menú de la barra inferior** que ya tenemos para crear el menú lateral (navigation drawer). Para hacerlo, necesitamos agregar **separadores** para organizar las acciones en secciones dentro del menú. Para ello, modificamos el archivo “menu\_main.xml” y crear un **separador** para la sección de ayuda. Esto nos permitirá agrupar elementos en un submenú específico.

Añadiremos un nuevo elemento con el título correspondiente, que en este caso será “Support”. Este elemento creará una sección donde incluiremos un submenú con la opción de ayuda:

<item android:title="Support">

<menu>

<item

android:id="@+id/helpFragment"

android:icon="@android:drawable/ic\_menu\_help"

android:title="Help"

app:showAsAction="always"></item>

</menu>

</item>

Con esta estructura, estamos creando un separador que contiene un submenú con la opción “Help”.

En cuanto a los otros **elementos** dentro de este menú, vamos a agregarlos a un **grupo** para que podamos aplicar propiedades comunes a todos ellos. En este caso, aplicaremos un comportamiento que permite resaltar el elemento seleccionado. Para lograr esto, crearemos un grupo con la etiqueta “group”, incluiremos los diferentes elementos y estableceremos la propiedad “checkableBehavior” en “single”:

<group android:checkableBehavior="single">

<item

android:id="@+id/inboxFragment"

android:icon="@android:drawable/ic\_dialog\_email"

android:title="Inbox"

app:showAsAction="always"></item>

<item

android:id="@+id/sentItemsFragment"

android:icon="@android:drawable/ic\_menu\_send"

android:title="Sent Items"

app:showAsAction="always"></item>

</group>

Este grupo nos permite aplicar un comportamiento de **selección única**, lo que significa que solo un elemento a la vez se resaltará como seleccionado en el menú lateral.

Drawer Layout

Vamos a configurar nuestro diseño principal en el archivo activity\_main.xml El elemento raíz de este diseño será un “DrawerLayout” para implementar el navigation drawer:

<?xml version="1.0" encoding="utf-8"?>

<androidx.drawerlayout.widget.DrawerLayout

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

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:id="@+id/drawer\_layout" >

1. **Diseño de la pantalla principal:** Conservaremos el “LinearLayout” existente con los elementos de la pantalla principal. Sin embargo, eliminaremos el menú inferior que ya no necesitamos.

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<!-- Material Toolbar -->

<com.google.android.material.appbar.MaterialToolbar

android:id="@+id/toolbar"

android:layout\_width="match\_parent"

android:layout\_height="?attr/actionBarSize"

style="@style/Widget.MaterialComponents.Toolbar.Primary" />

<!-- Fragment Container -->

<androidx.fragment.app.FragmentContainerView

android:id="@+id/nav\_host\_fragment"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:name="androidx.navigation.fragment.NavHostFragment"

app:navGraph="@navigation/nav\_graph"

android:layout\_weight="1"

app:defaultNavHost="true" />

<!-- Bottom Navigation Bar

<com.google.android.material.bottomnavigation.BottomNavigationView

android:id="@+id/bottom\_navigation"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

app:menu="@menu/menu\_main" /> -->

</LinearLayout>

1. **NavigationView para el panel de navegación:** Agregamos un “NavigationView” que contendrá el panel lateral de navegación. Le damos un ancho ajustado al contenido y una altura que ocupe toda la pantalla. Además, utilizamos la propiedad “layout\_gravity” para posicionarlo en la parte izquierda (start) de la pantalla (ten en cuenta que dependerá de la configuración de idioma). También especificamos un identificador, la cabecera que creamos previamente y el menú que configuramos.

<com.google.android.material.navigation.NavigationView

android:layout\_width="wrap\_content"

android:layout\_height="match\_parent"

android:layout\_gravity="start"

android:id="@+id/nav\_view"

app:headerLayout="@layout/nav\_header"

app:menu="@menu/menu\_main" />

Con estos cambios, hemos integrado el NavigationView en el DrawerLayout. Sin embargo, para ver el menú lateral en la aplicación, debemos enlazarlo con la barra de herramientas (toolbar) y agregar la funcionalidad de navegación.

Integrar en Toolbar

Para integrar el **panel lateral** con la barra de herramientas (toolbar), necesitamos realizar algunos ajustes en el código de la actividad principal (MainActivity):

1. Obtener una referencia al “DrawerLayout” que hemos creado en el diseño de la actividad principal:

val drawerLayout = findViewById<DrawerLayout>(R.id.drawer\_layout)

1. Utilizar un método específico de la clase “AppBarConfiguration.Builder” para indicar que el “DrawerLayout” es parte de la configuración de la barra de herramientas:

val builder = AppBarConfiguration.Builder(navController.graph)

builder.setOpenableLayout(drawerLayout)

val appBarConfiguration = builder.build()

Con estos ajustes, cuando la barra de herramientas se configure, incluirá un **botón** que permitirá abrir y cerrar el panel lateral. Una vez que realices estos cambios, al ejecutar la aplicación, podrás comprobar que el panel lateral está presente en la interfaz, aunque aún no esté completamente funcional ya que las funciones en el panel lateral aún no están vinculadas con el sistema de navegación.

Enlazar Controlador

Para enlazar los elementos del panel lateral con el sistema de navegación, procedemos de manera similar a como lo hicimos con la barra de navegación inferior (BottomNavigationView). Tenemos el componente NavigationView en el archivo activity\_main.xml, que representa el panel lateral. Para enlazarlo, seguimos estos pasos:

1. Obtener una referencia al componente NavigationView en la actividad principal (MainActivity):

val navigationView = findViewById<NavigationView>(R.id.nav\_view)

1. Usar el método setupWithNavController para enlazar el NavigationView con el controlador de navegación:

navigationView.setupWithNavController(navController)

Una vez realizados estos pasos, los botones en el panel lateral estarán vinculados al sistema de navegación y funcionarán correctamente.

Al ejecutar la aplicación, podrás comprobar que los botones del panel lateral funcionan como se esperaba.

Practica

**Descripción de la Práctica:**

En esta práctica, crearás una aplicación de Android UD04\_2\_Space que permitirá a los usuarios ver elementos visuales como imágenes de galaxias, planetas y estrellas, junto con colores oscuros y acentos luminosos que representen el misterio del cosmos.

Fragmento 1: “Planetas del Sistema Solar”

Este fragmento mostrará información sobre los planetas que componen nuestro sistema solar. Incluirá características principales como tamaño, distancia al Sol, composición atmosférica y temperatura.

Fragmento 2: “Estrellas y Constelaciones”

Este fragmento estará dedicado a las estrellas y las constelaciones más conocidas. Incluirá: representación visual de algunas de las constelaciones más reconocidas, como Orión, la Osa Mayor, Casiopea, etc.

* **Optativo: Botón para ver Leyendas y mitología**: historias detrás de las constelaciones y cómo han sido interpretadas en diferentes culturas a lo largo de la historia.
* **Optativo: Botón Tips para observación**: recomendaciones sobre cómo y cuándo observar ciertas constelaciones, con sugerencias de aplicaciones para astronomía o telescopios básicos.

Fragmento 3: “Exploración de la Luna y Marte”

En este fragmento se presentará información sobre la exploración espacial de la Luna y Marte. Incluirá detalles sobre las misiones Apollo, la llegada del ser humano a la Luna y descubrimientos relevantes.

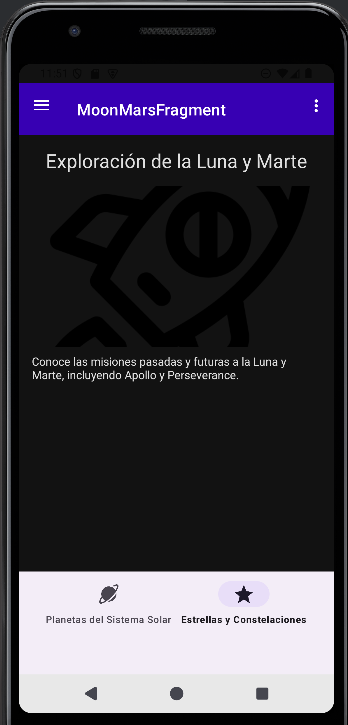
* **Misiones en Marte**: información sobre las misiones más importantes, como el rover Curiosity, Perseverance y las futuras misiones planificadas para colonización.
* **Optativo: Galería multimedia**: imágenes y videos de las misiones espaciales más relevantes.

Fragmento 4: “Tecnología Espacial y Misiones Futuras”

Este fragmento abordará las innovaciones tecnológicas que han hecho posible la exploración espacial y lo que está por venir. Incluirá detalles sobre el funcionamiento de cohetes, satélites y estaciones espaciales como la ISS.

Menús

Nuestro aplicación tendrá:

* **Toolbar y bottomBar**[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/5.practica/index.html#R-image-687172a35ce9f9dfcea44e44a67b198e)
* **Sidebar**[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud04/5.practica/index.html#R-image-4cf59d1749551b8a3524e590886d2a43)

**Requisitos Técnicos:**

* Cada fragmento proporcionará una experiencia educativa y visual atractiva para los usuarios interesados en la astronomía y la exploración del espacio.
* Deberás utilizar fragmentos para implementar cada una de las pantallas.
* La aplicación debe tener un diseño atractivo.
* Se deben utilizar menús.

**Consideraciones Adicionales:**

* Puedes utilizar recursos como imágenes o descripciones para cada temática de cuento.
* Asegúrate de que la aplicación sea fácil de usar y que los usuarios puedan seguir el flujo lógico de introducir su nombre, seleccionar una temática y leer su cuento personalizado.

Capitulo 5

UD05. Material Views

Resultados de avaliación

**RA1.** Aplica tecnoloxías de desenvolvemento para dispositivos móbiles, e avalía as súas características e as súas capacidades.

**RA2.** Desenvolve aplicacións para dispositivos móbiles, para o que analiza e emprega as tecnoloxías e as librarías específicas.

Criterios de avaliación

* CA1.8 Realizáronse modificacións sobre aplicacións existentes.
* CA1.9 Utilizáronse emuladores para comprobar o funcionamento das aplicacións.
* CA2.1 Xerouse a estrutura de clases necesaria para a aplicación.
* CA2.4 Utilizáronse as clases necesarias para a conexión e a comunicación con dispositivos sen fíos.
* CA2.5 Utilizáronse as clases necesarias para o intercambio de mensaxes de texto e multimedia.
* CA2.8 Realizáronse probas de interacción entre o usuario e a aplicación para mellorar as aplicacións desenvolvidas a partir de emuladores.
* CA2.9 Empaquetáronse e despregáronse as aplicacións desenvolvidas en dispositivos móbiles reais.
* CA2.10 Documentáronse os procesos necesarios para o desenvolvemento das aplicacións

BC1. Análise de tecnoloxías para desenvolvemento de aplicacións en dispositivos móbiles.

* Contidos
* Contornos integrados de traballo.
* Módulos para o desenvolvemento de aplicacións móbiles.
* Emuladores.
* Estrutura dunha aplicación para dispositivo móbil.
* Modificación de aplicacións existentes.
* Uso do contorno de execución do administrador de aplicacións.
* Ferramentas e fases de construción.
* Técnicas de animación e son.
* Comunicacións: clases asociadas. Tipos de conexións.
* Xestión da comunicación sen fíos.
* Envío e recepción de mensaxes de texto: seguridade e permisos.
* Envío e recepción de mensaxaría multimedia: sincronización de contido; seguridade e permisos.
* Manexo de conexións HTTP e HTTPS.
* Eventos da interface.
* Probas de interacción.
* Empaquetaxe e distribución.
* Documentación do desenvolvemento das aplicacións.

Crear proyecto

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
   * Selecciona “Start a new Android Studio project” en la pantalla de inicio.
   * Elige “Phone and Tablet” como tipo de dispositivo.
   * Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
   * En la siguiente pantalla, completa la información básica sobre el proyecto:
     + **Name**: Ingresa “UF1\_UD05\_1\_Pizz”.
     + **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
     + **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
     + **Language**: Selecciona “Kotlin” como lenguaje de programación.
     + **Minimum API level**: Selecciona “API 35: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
   * Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
   * Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad MainActivity.kt y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

Material

En la biblioteca estándar de Android y en **Material**, contamos con diversos componentes para mejorar nuestra interfaz de usuario. A continuación, exploraremos cómo utilizar algunos de ellos:

* **Toolbars** dinámicas que se ajustan al desplazamiento de la pantalla o se colapsan.
* **Radio buttons**, **checkboxes** y **chips** que facilitan la selección de opciones.
* **Botones de acción flotantes** (Floating Action Buttons o FAB’s).
* **Toasts** y **Snackbars** para mostrar mensajes emergentes.

Dependencia

Antes que nada, es crucial verificar si nuestro proyecto cuenta con la dependencia de Material agregada. En el archivo Gradle de la aplicación, asegurémonos de que la dependencia esté incluida de la siguiente manera:

implementation("com.google.android.material:material:1.10.0")

Fragmento

Hemos creado el fragmento OrderFragment y simplificado el archivo Kotlin para que solo contenga el método onCreateView:

class OrderFragment : Fragment() {

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

// Inflar el diseño para este fragmento

return inflater.inflate(R.layout.fragment\_order, container, false)

}

}

Además, hemos ajustado el diseño para que muestre únicamente un texto con la pregunta “¿Qué tipo de pizza deseas?”. Para lograrlo, hemos agregado una cadena:

<string name="order\_question">¿Qué tipo de pizza deseas?</string>

Y hemos modificado el diseño:

<?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"

tools:context=".OrderFragment"

android:orientation="vertical" >

<TextView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:text="@string/order\_question" />

</LinearLayout>

Pasos

1. **Añadiendo el Fragmento a la Actividad Principal:** Para integrar el fragmento en la actividad principal, creamos un LinearLayout que utiliza el componente FragmentContainerView.

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

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

xmlns:tools="http://schemas.android.com/tools"

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<androidx.fragment.app.FragmentContainerView

android:id="@+id/fragment\_container\_view"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:name="com.example.uf1\_ud05\_1\_pizza.OrderFragment"/>

</LinearLayout>

Ejecutamos la aplicación hasta este punto para confirmar que el fragmento se carga correctamente en el contenedor.

1. **Cambiando la Barra por Defecto:**

* Verificamos que estamos utilizando un tema sin la barra por defecto en ambos archivos “themes.xml”:

<style name="Base.Theme.UF1\_UD05\_1\_Pizza" parent="Theme.Material3.DayNight.NoActionBar">

* Creamos nuestra propia barra en el archivo del fragmento, asociándola específicamente al fragmento:

<com.google.android.material.appbar.MaterialToolbar

android:id="@+id/toolbar"

android:layout\_width="match\_parent"

android:layout\_height="?attr/actionBarSize"

style="@style/Widget.MaterialComponents.Toolbar.Primary" />

Ejecutamos la aplicación para notar que hay una barra presente, aunque aún no está definida como la barra por defecto.

1. **Toolbar como Barra Principal:** En el código del fragmento, recuperamos la toolbar y la establecemos como la barra principal:

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

// Inflar el diseño para este fragmento

val view = inflater.inflate(R.layout.fragment\_order, container, false)

val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)

(activity as AppCompatActivity).setSupportActionBar(toolbar)

return view

}

Con este código, aunque en un fragmento no tenemos acceso directo al método setSupportActionBar, podemos acceder a la actividad que contiene el fragmento para establecer la toolbar como la barra principal.

Scroll

En esta fase, deseamos implementar el **desplazamiento** en el fragmento, de modo que al hacer scroll, la toolbar también se desplace hacia abajo.

Para lograr que la toolbar se mueva al hacer scroll, necesitamos emplear un CoordinatorLayout, encargado de coordinar las animaciones entre las diversas vistas. Se incluirá un AppBarLayout, que posibilitará la animación de la toolbar, y un NestedScrollView, que permitirá que el contenido del diseño sea desplazable.

<CoordinatorLayout>

<AppBarLayout>

<CollapsingToolbarLayout>

<MaterialToolbar/>

</CollapsingToolbarLayout>

</AppBarLayout>

<NestedScrollView>

<LinearLayout> <!-- Solo puede haber un view -->

. . .

</LinearLayout>

</NestedScrollView>

</CoordinatorLayout>

Para implementar una barra colapsable, introduciremos un nuevo tipo de diseño dentro del AppBarLayout: el CollapsingToolbarLayout. Este, a su vez, contendrá la MaterialToolbar.

<CoordinatorLayout>

<AppBarLayout>

<CollapsingToolbarLayout>

<MaterialToolbar/>

</CollapsingToolbarLayout>

</AppBarLayout>

<NestedScrollView>

<LinearLayout> <!-- Solo puede haber un view -->

. . .

</LinearLayout>

</NestedScrollView>

</CoordinatorLayout>

1. **Cambiar a CoordinatorLayout:** Primero, actualizamos nuestro diseño reemplazando el LinearLayout por el CoordinatorLayout. Todo permanece igual, excepto la orientación, que es específica del LinearLayout.

<androidx.coordinatorlayout.widget.CoordinatorLayout

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"

tools:context=".OrderFragment">

1. **Añadir AppBarLayout:** Agregamos el diseño específico appBarLayout:

<com.google.android.material.appbar.AppBarLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:theme="@style/ThemeOverlay.MaterialComponents.Dark">

1. **Añadir Toolbar dentro del AppBarLayout:** Dentro de este diseño específico para la barra de aplicación, añadimos la toolbar para que responda al scroll:

<com.google.android.material.appbar.AppBarLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:theme="@style/ThemeOverlay.MaterialComponents.Dark">

<com.google.android.material.appbar.MaterialToolbar

android:id="@+id/toolbar"

android:layout\_width="match\_parent"

android:layout\_height="?attr/actionBarSize" />

</com.google.android.material.appbar.AppBarLayout>

Ya no necesitamos el estilo de la barra porque aplicaremos el tema al diseño AppBarLayout.

1. **Definir Propiedad de Scroll:** Definimos una propiedad para especificar cómo reaccionará ante los scrolls:

app:layout\_scrollFlags="scroll|enterAlways"

“scroll” para permitir el desplazamiento y “enterAlways” para ocultar la barra superior al hacer scroll.

1. **Scroll en el Resto del Fragmento:**

Además de la toolbar, queremos que el resto del contenido del fragmento haga scroll. Por lo tanto, añadimos los demás componentes dentro del elemento NestedScrollView. También le añadimos la propiedad app:layout\_behavior="@string/appbar\_scrolling\_view\_behavior" para ajustar su posición en función de si la toolbar es visible o no.

<androidx.core.widget.NestedScrollView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

app:layout\_behavior="@string/appbar\_scrolling\_view\_behavior">

<TextView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:text="@string/order\_question" />

</androidx.core.widget.NestedScrollView>

Al ejecutar la aplicación, podemos observar que al pulsar sobre el texto, hacemos scroll, ocultando la toolbar.

# Collapsing bar

Vamos a ajustar el comportamiento de desplazamiento de la barra de aplicación para que, en lugar de hacer scroll, se colapse. Para lograr esto, agregamos otro diseño del tipo CollapsingToolbarLayout dentro del diseño de AppBarLayout. A través de la propiedad layout\_behavior, indicamos que este diseño debe colapsar:

<com.google.android.material.appbar.AppBarLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:theme="@style/ThemeOverlay.MaterialComponents.Dark">

<com.google.android.material.appbar.CollapsingToolbarLayout

android:layout\_width="match\_parent"

android:layout\_height="300dp"

app:layout\_scrollFlags="scroll|exitUntilCollapsed">

<com.google.android.material.appbar.MaterialToolbar

android:id="@+id/toolbar"

android:layout\_width="match\_parent"

android:layout\_height="?attr/actionBarSize"

app:layout\_collapseMode="pin" />

</com.google.android.material.appbar.CollapsingToolbarLayout>

</com.google.android.material.appbar.AppBarLayout>

Ahora es el CollapsingToolbarLayout el que responde al comportamiento de desplazamiento, por lo que la propiedad layout\_scrollFlags ya no es necesaria en la MaterialToolbar. No obstante, vamos a indicar con otra propiedad cómo debe colapsar. En este caso, si tenemos un título, botones, texto, etc., colapsará pero dejará visible esos elementos:

<com.google.android.material.appbar.MaterialToolbar

android:id="@+id/toolbar"

android:layout\_width="match\_parent"

android:layout\_height="?attr/actionBarSize"

app:layout\_collapseMode="pin" />

Al ejecutar la aplicación, al pulsar sobre el texto, observaremos que hacemos scroll y la toolbar se colapsa, pero siempre deja visible el texto.

# Imagen

Vamos a incorporar una [imagen](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud05/6.imagen/restaurant.webp) a la toolbar. Inicialmente, la añadimos a la carpeta “drawable” y luego insertamos un componente ImageView dentro de nuestro CollapsingBarToolBar.

Al utilizar el componente parallax, podemos tener varios elementos dentro de nuestra toolbar y, al colapsar, experimentarán una animación específica, creando un estilo distintivo.

# Color

Adicionalmente, podemos incluir una propiedad para que, al colapsar, se aplique un color específico a la barra de título:

app:contentScrim="?attr/colorPrimary"

Al ejecutar la aplicación, observamos el siguiente resultado:

Capitulo 8

Radio Buttons

Los botones de radio, también conocidos como [radio buttons](https://developer.android.com/guide/topics/ui/controls/radiobutton?hl=es-419), posibilitan la elección de una opción de entre varias alternativas mutuamente excluyentes.

* Para implementar un botón de radio, se utilizará la vista RadioButton.
* El método checkRadioButtonId proporciona acceso al ID del RadioButton seleccionado (devuelve -1 si no se ha seleccionado ninguno).
* Dado que representan opciones excluyentes, es necesario agruparlos dentro de un RadioGroup. Esto garantiza que la selección de uno desactive automáticamente los demás.

Incorporar al proyecto

Para incorporar nuevos componentes a nuestro proyecto, es necesario realizarlo dentro del elemento NestedScrollView. No obstante, dado que solo se permite una vista dentro de este elemento, y ya hemos incluido un TextView, ahora deseamos agregar más componentes. Para lograr esto, es preciso insertar un diseño (layout) dentro de él, que sirva como raíz para los demás elementos que planeamos añadir. El código modificado es el siguiente:

<androidx.core.widget.NestedScrollView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

app:layout\_behavior="@string/appbar\_scrolling\_view\_behavior">

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:padding="16dp"

android:orientation="vertical">

<TextView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:text="@string/order\_question" />

<!-- Aquí puedes añadir más componentes según sea necesario -->

<!-- Por ejemplo: -->

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Mi Botón" />

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="Mi Campo de Texto" />

<!-- Agrega otros componentes aquí según tus requisitos -->

</LinearLayout>

</androidx.core.widget.NestedScrollView>

Es importante destacar que este ajuste no debería afectar la apariencia de nuestra aplicación.

Radio Group

Dado que los botones de radio son mutuamente excluyentes, es necesario crear un grupo que los agrupe:

<RadioGroup

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/group\_pizza\_type">

<!-- Aquí se agregarán los botones de radio según sea necesario -->

</RadioGroup>

Radio Button

Dentro de este conjunto, incorporamos los distintos botones de radio:

<RadioGroup

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/group\_pizza\_type">

<RadioButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/radio\_margarita"

android:text="Margarita" />

<RadioButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/radio\_calzone"

android:text="Calzone" />

</RadioGroup>

Cuando ninguno de los botones está seleccionado, el ID devolverá -1. Si seleccionamos alguno de ellos, obtendremos esos IDs.

Chips

Los [Chips](https://developer.android.com/training/wearables/compose/chips?hl=es-419) son elementos interactivos que posibilitan tanto selecciones como acciones:

* Pueden contener hasta dos etiquetas (principal y secundaria) y un icono, y también admiten una imagen de fondo en el caso de Chips de imagen.
* Se implementan mediante la vista Chip y, aunque operan de manera independiente, es posible agruparlos visualmente utilizando ChipGroup.
* Su estado de selección puede ser verificado a través del método isChecked.

Incorporar al proyecto

Vamos a incorporar dos chips a nuestro proyecto, ofreciendo al usuario la posibilidad de seleccionar entre dos opciones no excluyentes. Dentro del LinearLayout, añadimos ambos componentes:

<com.google.android.material.chip.Chip

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Parmesano"

android:id="@+id/chip\_parmesano"

style="@style/Widget.MaterialComponents.Chip.Choice" />

<com.google.android.material.chip.Chip

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/chip\_tomate\_cherry"

android:text="Tomate Cherry"

style="@style/Widget.MaterialComponents.Chip.Choice" />

Verificamos que se posicionan uno debajo del otro debido al diseño del LinearLayout. Para aplicar estilos específicos, podemos agruparlos mediante un ChipGroup:

<com.google.android.material.chip.ChipGroup

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content">

<com.google.android.material.chip.Chip ... />

<com.google.android.material.chip.Chip ... />

</com.google.android.material.chip.ChipGroup>

Una vez incluidos los dos chips dentro del ChipGroup, observamos que ahora se presentan horizontalmente.

Estos chips representan una evolución de los checkboxes, permitiendo comportamientos adicionales. Si modificamos la propiedad de selección:

style="@style/Widget.MaterialComponents.Chip.Action"

Funcionarán como botones, desencadenando eventos que podemos controlar desde nuestro código.

En el caso de aplicar otro estilo, como el de filtro:

style="@style/Widget.MaterialComponents.Chip.Filter"

Al ejecutar nuestra aplicación, notamos que la apariencia ha cambiado.

FABs

Los [botones de acción flotantes](https://m3.material.io/components/floating-action-button/overview) son elementos circulares cuya función principal es activar la acción primaria en la interfaz de usuario de tu aplicación.

* Los íconos deben ser claros, ya que no cuentan con etiquetas explícitas.
* Mantienen su presencia en la pantalla incluso cuando se realiza un desplazamiento.
* A lo largo de las distintas versiones del sistema Material Design, su apariencia ha experimentado modificaciones.

Para incorporar estos elementos a nuestro proyecto, los añadiremos fuera del NestedScrollView, ya que no forman parte de los elementos básicos de nuestro sistema. Por lo tanto, nos posicionamos en el elemento raíz CoordinatorLayout y añadimos el siguiente componente:

<com.google.android.material.floatingactionbutton.FloatingActionButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/fab"

android:layout\_margin="16dp"

android:src="@android:drawable/ic\_menu\_send"

android:layout\_gravity="bottom|end" />

Integración en la Barra de Herramientas

Vamos a realizar otra integración, esta vez dentro de la barra de herramientas.

1. Para incorporar este botón en la barra de herramientas, lo primero que debemos hacer es asignarle un identificador a esa barra:
2. <com.google.android.material.appbar.CollapsingToolbarLayout
3. android:layout\_width="match\_parent"
4. android:layout\_height="300dp"
5. app:layout\_scrollFlags="scroll|exitUntilCollapsed"
6. app:contentScrim="?attr/colorPrimary"

android:id="@+id/collapsing\_toolbar">

1. Copiamos el código del botón flotante anterior (fab) y le añadimos dos propiedades adicionales: layout\_anchor, donde indicamos dónde vamos a anclar este botón flotante (en este caso, a la barra de herramientas), y layout\_anchorGravity, donde especificamos en qué parte de la barra de herramientas queremos que se posicione:
2. <com.google.android.material.floatingactionbutton.FloatingActionButton
3. android:layout\_width="wrap\_content"
4. android:layout\_height="wrap\_content"
5. android:id="@+id/fab2"
6. android:layout\_margin="16dp"
7. android:src="@android:drawable/ic\_menu\_help"
8. android:layout\_gravity="bottom|end"
9. app:layout\_anchor="@id/collapsing\_toolbar"

app:layout\_anchorGravity="bottom|end" />

Toast y Snackbar

Los Toasts y Snackbars son herramientas eficaces para proporcionar al usuario información o avisos de manera concisa en una ventana emergente que desaparece automáticamente después de un tiempo.

Toasts

Obtendremos una referencia a nuestro botón:

```kotlin

val fab = view.findViewById<FloatingActionButton>(R.id.fab)

fab.setOnClickListener {

val pizzaGroup = view.findViewById<RadioGroup>(R.id.group\_pizza\_type)

val pizzaType = pizzaGroup.checkedRadioButtonId

if (pizzaType == -1) {

val msg = "Debes seleccionar un tipo de pizza"

Toast.makeText(activity, msg, Toast.LENGTH\_LONG).show()

} else {

// Lógica adicional, si es necesario

}

}

```

Si no se ha seleccionado ningún tipo de pizza, generamos un Toast con un mensaje.

![chips](ud4-toast.gif)

Snackbar

Añadir un mensaje de Snackbar es similar a Toast:

```kotlin

Snackbar.make(fab, msg, Snackbar.LENGTH\_SHORT).show()

```

Completamos la lógica escribiendo un mensaje con el tipo de pizza seleccionado y los elementos adicionales que queremos:

```kotlin

var msg = ""

if (pizzaType == -1) {

msg = "Debes seleccionar un tipo de pizza"

// Toast.makeText(activity, msg, Toast.LENGTH\_LONG).show()

} else {

msg = "Has seleccionado una pizza "

// Tipo de pizza

msg += when (pizzaType) {

R.id.radio\_margarita -> "Margarita"

else -> "Calzone"

}

// Extras

var parmesano = view.findViewById<Chip>(R.id.chip\_parmesano)

if (parmesano.isChecked) msg += " Parmesano"

var tomate = view.findViewById<Chip>(R.id.chip\_tomate\_cherry)

if (tomate.isChecked) msg += " Tomate Cherry"

}

Snackbar.make(fab, msg, Snackbar.LENGTH\_SHORT).show()

```

![chips](ud4-total.gif)

Estas herramientas proporcionan una manera efectiva de comunicarse con el usuario de forma breve y no intrusiva. Otra opción sería recuperar el elemento directamente de la vista y obtener el texto mostrado: ```kotlin

var msn = “Debes seleccionar un tipo de pizza”.plus(view.findViewById(pizzaType).text) /\* msn += when(burguerType){ R.id.radio\_american -> getString(R.string.american) R.id.radio\_vegan -> getString(R.string.vegan) R.id.radio\_chicken -> getString(R.string.chicken) else -> “Error” }\*/ val snackbar = Snackbar.make(fabSend, msn, Snackbar.LENGTH\_SHORT) .setAction(“Undo”){

}.show()

## Acción SnackBar

Estos Snackbars también nos brindan la posibilidad de agregar acciones. Esto se logra antes de mostrar el SnackBar. A continuación, se describen los pasos:

1. Primero, obtenemos una referencia al SnackBar:

```kotlin

val snackbar = Snackbar.make(fab, msg, Snackbar.LENGTH\_SHORT)

1. Luego, añadimos una acción “Undo”:
2. val snackbar = Snackbar.make(fab, msg, Snackbar.LENGTH\_SHORT)
3. snackbar.setAction("Undo") {
4. // Acciones a realizar cuando se selecciona "Undo"
5. }

snackbar.show()

Estos pasos nos permiten personalizar el SnackBar con una acción específica que el usuario puede ejecutar, en este caso, deshacer la acción previa.

UD06. MAD (Modern Android Development)

Resultados de avaliación

**RA1.** Aplica tecnoloxías de desenvolvemento para dispositivos móbiles, e avalía as súas características e as súas capacidades.

**RA2.** Desenvolve aplicacións para dispositivos móbiles, para o que analiza e emprega as tecnoloxías e as librarías específicas.

Criterios de avaliación

* CA1.8 Realizáronse modificacións sobre aplicacións existentes.
* CA1.9 Utilizáronse emuladores para comprobar o funcionamento das aplicacións.
* CA2.1 Xerouse a estrutura de clases necesaria para a aplicación.
* CA2.4 Utilizáronse as clases necesarias para a conexión e a comunicación con dispositivos sen fíos.
* CA2.5 Utilizáronse as clases necesarias para o intercambio de mensaxes de texto e multimedia.
* CA2.8 Realizáronse probas de interacción entre o usuario e a aplicación para mellorar as aplicacións desenvolvidas a partir de emuladores.
* CA2.9 Empaquetáronse e despregáronse as aplicacións desenvolvidas en dispositivos móbiles reais.
* CA2.10 Documentáronse os procesos necesarios para o desenvolvemento das aplicacións

BC1. Análise de tecnoloxías para desenvolvemento de aplicacións en dispositivos móbiles.

* Contidos
* Contornos integrados de traballo.
* Módulos para o desenvolvemento de aplicacións móbiles.
* Emuladores.
* Estrutura dunha aplicación para dispositivo móbil.
* Modificación de aplicacións existentes.
* Uso do contorno de execución do administrador de aplicacións.
* Ferramentas e fases de construción.
* Técnicas de animación e son.
* Comunicacións: clases asociadas. Tipos de conexións.
* Xestión da comunicación sen fíos.
* Envío e recepción de mensaxes de texto: seguridade e permisos.
* Envío e recepción de mensaxaría multimedia: sincronización de contido; seguridade e permisos.

# Introducción

A lo largo de los años, las arquitecturas típicas de las aplicaciones Android han experimentado una evolución significativa. En el pasado, los modelos constructivos se basaban en patrones como MVC (Model-View-Controller) o MVP (Model-View-Presenter). Sin embargo, con la introducción de nuevas soluciones y tecnologías, los proyectos más recientes han adoptado patrones como MVVM (Model-View-ViewModel) o MVI (Model-View-Intent).

En este contexto de cambio, es fundamental explorar algunas de estas tecnologías incluidas en Jetpack, una colección de bibliotecas y herramientas recomendadas por Android:

* **View Binding:** Esta tecnología permite vincular los componentes de la interfaz de usuario con el código de manera más segura y eficiente, reemplazando la necesidad de usar findViewById.
* **View Models:** Los ViewModel son componentes que ayudan a gestionar y mantener datos relacionados con la interfaz de usuario, lo que facilita la gestión de ciclos de vida y la separación de preocupaciones.
* **Live Data:** LiveData es una clase diseñada para almacenar y observar datos de manera reactiva, lo que simplifica la actualización de la interfaz de usuario cuando cambian los datos subyacentes.
* **Data Binding, Room databases, …:** Estas tecnologías amplían aún más las capacidades de desarrollo, permitiendo la vinculación de datos directamente a la interfaz de usuario, gestionando bases de datos locales con facilidad y mejorando la calidad y la eficiencia del código en general.

La adopción de estas tecnologías y patrones arquitectónicos ha revolucionado la forma en que se desarrollan aplicaciones Android, mejorando la organización del código y la experiencia del usuario.

# Model-View-ViewModel

La arquitectura MVVM (Model-View-ViewModel) es un patrón de diseño ampliamente utilizado en el desarrollo de aplicaciones Android. Su objetivo principal es separar las preocupaciones y mejorar la organización del código en las aplicaciones, lo que facilita el mantenimiento y la escalabilidad. Aquí te explico cómo funciona:

1. **Model (Modelo):** El modelo representa los datos y la lógica de negocio de tu aplicación. Puede incluir clases que representen los objetos de datos, así como componentes para acceder y administrar esos datos, como bases de datos, servicios web o repositorios. El modelo no tiene conocimiento de la interfaz de usuario.
2. **View (Vista):** La vista es la capa de la interfaz de usuario que se encarga de mostrar los datos y permitir la interacción del usuario. En Android, la vista suele estar representada por las actividades, fragmentos y diseños de interfaz de usuario.
3. **ViewModel:** El ViewModel actúa como intermediario entre el Modelo y la Vista. Contiene la lógica de presentación y se encarga de preparar y proporcionar los datos necesarios para que la vista los muestre. También maneja las interacciones del usuario y cualquier lógica relacionada con la vista, como validaciones y formateo de datos.

El flujo de datos en una arquitectura MVVM típica es el siguiente:

1. El **ViewModel** solicita datos al Modelo (por ejemplo, a través de un repositorio) y los almacena en propiedades observables.
2. La Vista se suscribe a estas propiedades observables del ViewModel.
3. Cuando los datos en el ViewModel cambian, se notifica a la Vista y se actualiza automáticamente la interfaz de usuario.
4. Cuando el usuario interactúa con la interfaz de usuario, la Vista envía eventos al ViewModel para que procese acciones, como guardar datos o realizar operaciones adicionales.

Una característica importante de MVVM es la capacidad de utilizar la vinculación de datos (data binding) para que los cambios en el ViewModel se reflejen automáticamente en la Vista, y viceversa. Esto simplifica la actualización de la interfaz de usuario y reduce la necesidad de código manual para sincronizar datos y vistas.

En Android, MVVM se implementa comúnmente con el uso de bibliotecas como LiveData para la comunicación entre el ViewModel y la Vista, y Data Binding para la vinculación de datos. Estas herramientas ayudan a simplificar la implementación de la arquitectura MVVM en aplicaciones Android, mejorando la organización y mantenimiento del código.

# View Binding

La implementación de View Binding en Android simplifica la escritura de código que interactúa con las vistas de la interfaz de usuario de una manera altamente eficiente y segura. Una vez habilitada, esta característica genera una clase de vinculación (binding class) para cada layout XML, estableciendo un mapeo directo con cada vista dentro del diseño. Esta funcionalidad se aplica tanto a las actividades como a los fragmentos en una aplicación.

Una de las ventajas más notables de View Binding es su capacidad para eliminar la necesidad de usar findViewById, lo que conlleva beneficios significativos:

* **Eficiencia:** Al evitar la búsqueda a través de la jerarquía de vistas en tiempo de ejecución, el código resulta más eficiente en términos de rendimiento, ya que Android no necesita realizar búsquedas innecesarias.
* **Seguridad:** A diferencia de findViewById, View Binding es type-safe. Elimina la necesidad de realizar casting y garantiza que las referencias sean seguras y coincidan con las vistas correctas. Además, elimina el riesgo de referencias nulas devueltas por el método en caso de que el ID suministrado sea inválido, ya que los problemas se detectan en tiempo de compilación, lo que mejora la robustez del código.

Si deseas obtener más información sobre View Binding, puedes consultar la documentación oficial en [este enlace](https://developer.android.com/topic/libraries/view-binding?hl=es-419).

# Dependencia

Para habilitar View Binding en tu proyecto, es necesario realizar un ajuste en el archivo build.gradle de la aplicación de la siguiente manera:

android {

. . .

buildFeatures {

viewBinding true

}

}

Una vez habilitado, View Binding generará **automáticamente** una clase de vinculación para cada actividad o fragmento. El nombre de esta clase se forma tomando el nombre de la actividad o fragmento al que está asociada, al cual se le agrega “**Binding**”. Este objeto de vinculación se encarga de mapear el diseño de la interfaz de usuario.

Para acceder a este objeto de vinculación en tus componentes, puedes seguir estos pasos:

* En el caso de actividades, define una propiedad que se inicializará en el método onCreate.
* Para fragmentos, debido a que estos pueden persistir más allá de la vida de sus vistas, crea una propiedad que se inicialice en onCreateView y se anule en onDestroyView.

Esto proporciona una manera segura y eficiente de interactuar con los elementos de la interfaz de usuario en tus actividades y fragmentos.

Si deseas obtener información adicional sobre View Binding, puedes consultar la documentación oficial en [este enlace](https://developer.android.com/topic/libraries/view-binding?hl=es-419).

View Binding en el Proyecto Chronos

Para actualizar la aplicación Chronos y utilizar View Binding, sigue estos pasos:

1. Activar Componente en el build.gradle de la aplicación:

Añade la siguiente configuración en el bloque buildFeatures:

android {

...

buildFeatures {

viewBinding = true

}

}

Guarda el archivo y actualiza el proyecto.

2. Implementar View Binding en la Actividad Principal:

// Importa la clase generada para el View Binding

import com.example.yourpackage.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

// Variable para almacenar la referencia del View Binding

private lateinit var binding: ActivityMainBinding

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

// Inflar el diseño utilizando View Binding

binding = ActivityMainBinding.inflate(layoutInflater)

val view = binding.root

setContentView(view)

// Acceso a los elementos de la interfaz mediante el objeto binding

val chrono = binding.chrTemporizador

val btnStart = binding.btnStart

val btnPause = binding.btnPause

// Resto del código...

}

}

Con estos cambios, ahora puedes acceder directamente a los elementos de la interfaz a través del objeto binding, eliminando la necesidad de usar findViewById y simplificando el código.

La implementación de View Binding hace que el código sea más claro y menos propenso a errores al proporcionar referencias seguras y directas a los elementos de la interfaz de usuario.

View Binding en el Proyecto Pizza App

En esta aplicación, aprenderemos a activar y utilizar un componente en un fragmento. Si recordamos, esta aplicación fue creada en base a un fragmento.

Activar Componente

Primero, activaremos el componente. Para ello, vamos al archivo build.gradle de la aplicación y añadimos:

buildFeatures {

viewBinding = true

}

Con esto, hemos completado la activación. Ahora, solo nos falta actualizar el proyecto:

sync now

View Binding en Fragmento

En este caso, declararemos una variable binding para almacenar los recursos y será de la clase FragmentOrderBinding:

private lateinit var binding: FragmentOrderBinding

Sin embargo, con los fragmentos, surge un problema, ya que el ciclo de vida de un fragmento puede diferir de los elementos que tiene en su diseño. En este caso, los elementos pueden destruirse, pero el fragmento puede seguir existiendo, lo que impediría acceder a los mismos.

Para resolver esto, crearemos otra variable (que comenzará con \_ para diferenciarla) que aceptará nulos:

private var \_binding: FragmentOrderBinding? = null

private val binding: FragmentOrderBinding

get() = \_binding!!

Ahora, indicaremos que si se destruye el fragmento, esa variable sea nula:

override fun onDestroyView() {

super.onDestroyView()

\_binding = null

}

Luego, recuperaremos la referencia al objeto y obtendremos una referencia al padre:

\_binding = FragmentOrderBinding.inflate(inflater, container, false)

val view = binding.root

Con esto, podemos eliminar todas las referencias a los elementos utilizando la variable binding. Por ejemplo:

binding.chipParmesano.isChecked

# View Model

La clase ViewModel actúa como un contenedor que encapsula el estado y la lógica a nivel de pantalla en una aplicación. Su función principal es **exponer** el estado de la aplicación a la interfaz de usuario (UI) y **encapsular** la lógica relacionada.

Una de las principales **ventajas** de utilizar ViewModel radica en su capacidad para gestionar automáticamente la persistencia durante la navegación entre actividades y destinos de Navigation, así como en cambios de configuración, como rotaciones de pantalla.

Un ViewModel puede estar asociado a una actividad o fragmento, y su ciclo de vida está vinculado al ciclo de vida de ese componente. Además, permite que dos o más fragmentos compartan un mismo objeto ViewModel si está creado y asociado al contenedor principal (ya sea una actividad o un fragmento). Este mecanismo facilita la compartición de datos entre diferentes destinos en la aplicación.

Para generar un modelo, existen varias formas, siendo la más sencilla el uso de la función viewModels(). Esta función devuelve una propiedad delegada que proporciona acceso al modelo. Por defecto, su alcance será el del fragmento o actividad donde se declara. Por ejemplo:

class MyFragment : Fragment() {

val viewModel: MyViewModel by viewModels()

}

Si se desea modificar el alcance, se puede hacer mediante el parámetro ownerProducer:

class MyFragment : Fragment() {

val viewModel: MyViewModel by viewModels(

ownerProducer = { this.requireActivity() }

)

}

Esta flexibilidad en la creación y alcance de ViewModels proporciona una gestión eficiente del estado y la lógica de la aplicación en entornos Android. Para obtener más detalles, se puede consultar la documentación oficial en [este enlace](https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419).

Crear proyecto

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
   * Selecciona “Start a new Android Studio project” en la pantalla de inicio.
   * Elige “Phone and Tablet” como tipo de dispositivo.
   * Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
   * En la siguiente pantalla, completa la información básica sobre el proyecto:
     + **Name**: Ingresa “UF1\_UD06\_3\_GuessGame”.
     + **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
     + **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
     + **Language**: Selecciona “Kotlin” como lenguaje de programación.
     + **Minimum API level**: Selecciona “API 35: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
   * Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
   * Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad MainActivity.kt y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

Dependencia

Activar View Binding

Para habilitar View Binding en tu proyecto, realiza el siguiente ajuste en el archivo build.gradle de la aplicación:

android {

. . .

buildFeatures {

viewBinding = true

}

}

Asegúrate de sincronizar los cambios para aplicar la configuración.

Dependencia para Crear Fragmentos

Para la creación de fragmentos, añade la siguiente dependencia en tu archivo build.gradle:

implementation("androidx.navigation:navigation-fragment-ktx:2.7.5")

Después de agregar la dependencia, realiza la sincronización para incorporar los cambios en tu proyecto.

Game Fragment

Vamos a crear un nuevo fragmento asociado a la pantalla de “Game”.

Layout del Fragmento

Actualizamos el diseño de nuestro fragmento. A continuación se muestra el nuevo XML del layout:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

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

xmlns:tools="http://schemas.android.com/tools"

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

android:layout\_height="match\_parent"

android:layout\_width="match\_parent"

android:orientation="vertical"

android:padding="16dp">

<TextView

android:id="@+id/txt\_word"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_gravity="center"

android:letterSpacing="0.2"

android:textSize="35sp"/>

<!-- Cuantas vidas nos quedan -->

<TextView

android:id="@+id/txt\_lives"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_marginTop="8dp"

android:letterSpacing="0.2"

android:textSize="16sp"/>

<!-- Introducir letras -->

<EditText

android:id="@+id/txt\_guess"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:inputType="text"

android:maxLength="1"

android:layout\_marginTop="8dp"

android:textSize="16sp"

android:gravity="center"

android:hint="Introduce una letra"/>

<!-- Boton siguiente -->

<Button

android:id="@+id/button\_next"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Prueba!"

android:layout\_gravity="center"

android:layout\_marginTop="8dp" />

</LinearLayout>

Código del Fragmento

Actualizamos el código del fragmento, eliminamos el código innecesario e incorporamos la propiedad binding:

class GameFragment : Fragment() {

private var \_binding: FragmentGameBinding? = null

private val binding get() = \_binding!!

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

\_binding = FragmentGameBinding.inflate(inflater, container, false)

val view = binding.root

return view

}

override fun onDestroyView() {

super.onDestroyView()

\_binding = null

}

}

Con estos cambios, hemos definido las propiedades del viewBinding para acceder a las propiedades del layout. Además, se ha simplificado el código del fragmento eliminando las partes innecesarias.

Result Fragment

Vamos a crear un nuevo fragmento asociado a la pantalla de “Result”.

Layout del Fragmento

Actualizamos el diseño de nuestro nuevo fragmento. A continuación, se presenta el nuevo XML del layout:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

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

xmlns:tools="http://schemas.android.com/tools"

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

android:layout\_height="match\_parent"

android:layout\_width="match\_parent"

android:orientation="vertical"

android:padding="16dp">

<TextView

android:id="@+id/txt\_result"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_gravity="center"

android:gravity="center"

android:textSize="28sp"/>

<!-- Boton Reiniciar -->

<Button

android:id="@+id/button\_new\_game"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Nueva Partida"

android:layout\_gravity="center"

android:layout\_marginTop="8dp" />

</LinearLayout>

Código del Fragmento

Actualizamos el código del fragmento y eliminamos el código innecesario:

package com.example.ud06\_3\_guessgame

import android.os.Bundle

import androidx.fragment.app.Fragment

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import com.example.ud06\_3\_guessgame.databinding.FragmentResultBinding

class ResultFragment : Fragment() {

private var \_binding: FragmentResultBinding? = null

private val binding get() = \_binding!!

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

\_binding = FragmentResultBinding.inflate(inflater, container, false)

return binding.root

}

override fun onDestroyView() {

super.onDestroyView()

\_binding = null

}

}

Con estos cambios, hemos definido las propiedades del viewBinding para acceder a las propiedades del layout. Además, se ha simplificado el código del fragmento eliminando las partes innecesarias.

# Grafo de Navegación

Generaremos un grafo de navegación denominado “nav\_graph.xml”, en el cual incorporaremos ambos fragmentos y estableceremos las rutas correspondientes:

Posteriormente, ajustaremos las propiedades de ambas rutas, especialmente la propiedad “PopUpTo”, para garantizar que no existan fragmentos previos al realizar la navegación:

# MainActivity

Vamos a realizar ajustes en la actividad principal para incorporar un contenedor de fragmentos. A continuación, se presenta el nuevo diseño XML:

<?xml version="1.0" encoding="utf-8"?>

<androidx.fragment.app.FragmentContainerView

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

xmlns:tools="http://schemas.android.com/tools"

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:id="@+id/fragment\_container\_view"

tools:context=".MainActivity"

android:name="androidx.navigation.fragment.NavHostFragment"

app:navGraph="@navigation/nav\_graph"

app:defaultNavHost="true"

>

</androidx.fragment.app.FragmentContainerView>

Con estos cambios, no será necesario realizar ninguna otra modificación en la actividad principal, ya que nuestro primer fragmento se cargará automáticamente.

# Game Model

Para implementar un ViewModel, debemos crear una clase llamada GameViewModel que herede de la clase ViewModel:

package com.example.ud06\_3\_guessgame

import androidx.lifecycle.ViewModel

class GameViewModel : ViewModel() {

}

Para realizar una prueba, generaremos una lista de palabras y seleccionaremos una palabra al azar de la lista, transformándola a mayúsculas:

val words = listOf("Android", "Fragment", "Kotlin", "Model")

var secretWord = words.random().uppercase()

Desde el fragmento, incluiremos una referencia a esta nueva clase:

val model: GameViewModel by viewModels()

La vida de este modelo está vinculada a la vida del fragmento, lo que nos permite acceder a cualquier método o propiedad del modelo desde el propio fragmento:

binding.buttonNext.setOnClickListener {

model.secretWord = "Prueba de modelo"

view.findNavController().navigate(R.id.action\_gameFragment\_to\_resultFragment)

}

Recuperaremos el modelo en el fragmento de Resultado y generaremos un Toast al hacer clic en el botón de nuevo juego:

binding.buttonNewGame.setOnClickListener {

Toast.makeText(activity, model.secretWord, Toast.LENGTH\_LONG).show()

}

En GameFragment, creamos un objeto a partir de nuestro modelo, modificamos el valor de una variable (secretWord) y navegamos al siguiente fragmento. Al avanzar al siguiente fragmento, mostramos con un Toast el valor de la variable.

Sin embargo, al ejecutar la aplicación, observamos que no se muestra la palabra que hemos definido en el fragmento.

# Compartir Modelo

Cuando creamos nuestro modelo, este queda vinculado al fragmento o a la actividad donde se crea, es decir, su alcance está limitado a ese contexto específico. Debido a esto, al navegar entre fragmentos, la información no se comparte automáticamente. Para abordar este desafío y asegurar que el modelo esté vivo durante toda la navegación entre fragmentos, necesitamos asociar su ciclo de vida al contenedor padre común de **ambos fragmentos.**

En este caso, indicamos que el propietario del modelo será la actividad que contiene el fragmento:

val model: GameViewModel by viewModels(

ownerProducer = { this.requireActivity() }s

)

Este enfoque se aplica en ambos fragmentos. Android verificará la existencia de esa actividad y, en caso de que no exista (como en la primera carga), creará el objeto. Sin embargo, al avanzar al siguiente fragmento y al intentar crear otro fragmento asociado a la misma actividad principal, Android detectará que el objeto ya existe. En consecuencia, almacenará la referencia al modelo existente en lugar de generar uno nuevo. Esto permite que los fragmentos subsiguientes accedan al modelo creado en el proceso anterior.

Esto resulta beneficioso para compartir información entre fragmentos, ya que la asociación ocurre antes de que se carguen los fragmentos. Además, podemos utilizar nuestras propias clases implementadas. Además, en situaciones como cambios de configuración (por ejemplo, rotación de la pantalla) donde la información podría perderse, no es necesario almacenarla en el bundle para recuperarla posteriormente.

# Lógica del Juego

En este momento vamos a implementar la lógica de nuestro juego. Para ello, nos dirigimos a la clase del **modelo** y creamos un método que permita especificar las palabras:

class GameViewModel: ViewModel() {

// Seleccionamos una palabra aleatoria de esta lista de palabras

val words = listOf("Android", "Fragment", "Kotlin", "Model")

var secretWord = words.random().uppercase()

// String que se mostrará en la pantalla (guiones y letras a medida que las vamos descubriendo)

var secretWordDisplay = ""

// Intentos del usuario. Caracteres que vaya probando el usuario.

var guesses = mutableListOf<Char>()

// Vidas

var lives = 8

init {

// Inicializamos la palabra con \_

secretWordDisplay = generateSecretWordDisplay()

}

// Genera la representación visual de la palabra oculta

fun generateSecretWordDisplay() =

// Recorremos cada uno de los caracteres de la palabra

secretWord.map {

// Si el caracter está en la lista, lo añadimos; sino, continuamos con \_

if (it in guesses) it

else '\_'

}.joinToString("")

// Realiza un intento de adivinanza por parte del usuario

fun makeGuess(guess: String){

if(guess.length > 0) {

// Extraemos la letra inicial (aunque solo nos pueden introducir un caracter)

val letter = guess.uppercase()[0]

// La añadimos a la lista de intentos

guesses.add(letter)

secretWordDisplay = generateSecretWordDisplay()

if(!secretWord.contains(letter)) lives -= 1

}

}

// Función para verificar si ganamos

fun win() = secretWord == secretWordDisplay

// Función para comprobar si nos quedan vidas

fun lost() = lives <= 0

}

Este modelo encapsula la lógica del juego y la representación visual de la palabra oculta, permitiendo que las interacciones del usuario se reflejen de manera coherente en la interfaz de usuario.

# Usar el Modelo

En este punto vamos a utilizar el modelo en nuestra actividad prinicipal:

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

// Inflate the layout for this fragment

\_binding = FragmentGameBinding.inflate(inflater, container, false)

val view = binding.root

//Inicializamos la palabra

updateScreen()

binding.buttonNext.setOnClickListener {

//model.secretWord = "Prueba de modelo"

if(binding.txtGuess.text.length>0){

//Comprobar la letra introducida

model.makeGuess(binding.txtGuess.text.toString())

//Actualizamos la pantalla

updateScreen()

//Si acertamos la palabra o nos quedamos sin vidas

if (model.win() || model.lost())

view.findNavController().navigate(R.id.action\_gameFragment\_to\_resultFragment)

}else{

//Sino se introduce ningún texto mostramos un aviso

Snackbar.make(view, "Introduce una letra", Snackbar.LENGTH\_SHORT).show()

}

}

return view

}

fun updateScreen(){

binding.txtWord.text = model.secretWordDisplay

binding.txtLives.text = "Te quedan ${model.lives} vidas"

binding.txtGuess.text = null

}

override fun onDestroyView() {

super.onDestroyView()

\_binding = null

}

Lógica Result Fragment

Vamos a generar la lógica del fragmento del resultado. Para ello vamos a nuestro viewModel:

Funciones en el Modelo

class GameViewModel : ViewModel() {

// ...

fun restart() {

guesses.clear()

lives = 8

secretWord = words.random().uppercase()

secretWordDisplay = generateSecretWordDisplay()

}

fun resultMessage() =

if (win()) "Ganaste!\n La palabra secreta era $secretWord"

else "Oops, perdiste!\n La palabra secreta era $secretWord"

// ...

}

Lógica del Fragmento:

// ...

binding.txtResult.text = model.resultMessage()

binding.buttonNewGame.setOnClickListener {

model.restart()

view.findNavController().navigate(R.id.action\_resultFragment\_to\_gameFragment)

// Toast.makeText(activity, model.secretWord, Toast.LENGTH\_LONG).show()

}

// ...

Estas funciones amplían la lógica del juego y proporcionan mensajes personalizados para mostrar al usuario después de completar o perder el juego.

1. **restart()**: Esta función reinicia el juego. Limpia la lista de intentos (guesses), restablece el número de vidas a 8, y selecciona una nueva palabra secreta aleatoria. Luego, genera la representación visual de la nueva palabra.
2. **resultMessage()**: Esta función devuelve un mensaje personalizado en función del resultado del juego. Si el jugador ha ganado, muestra un mensaje de victoria junto con la palabra secreta. Si el jugador ha perdido, muestra un mensaje de derrota con la palabra secreta.

Capitulo 5

# Live Data

## **LiveData en Android: Mejora de Observabilidad**

LiveData es una parte esencial de Jetpack que permite la declaración de datos observables, aplicando el patrón de diseño Observer. Este enfoque permite un seguimiento eficiente de las modificaciones en los datos, facilitando la actualización de la interfaz de usuario por parte de fragmentos y actividades.

**Ventajas clave de LiveData:**

1. **Actualización Automática de la UI:** LiveData permite que los fragmentos y actividades actualicen automáticamente su interfaz de usuario en respuesta a cambios en los datos. El observador se encarga de gestionar estas actualizaciones.
2. **Gestión del Ciclo de Vida:** LiveData gestiona automáticamente el ciclo de vida de los componentes, lo que significa que solo actualiza a los observadores activos. Esto previene problemas asociados con las fugas de memoria y actualizaciones innecesarias.
3. **Trabajo Conjunto con ViewModel:** LiveData y ViewModel suelen trabajar en conjunto. En el modelo (ViewModel), se establecen los atributos a observar mediante MutableLiveData.

Para obtener más detalles y ejemplos prácticos, puedes consultar la documentación oficial de LiveData en [este enlace](https://developer.android.com/topic/libraries/architecture/livedata?hl=es-419).

Vamos a mejorar la observabilidad de las variables lives y secretWordDisplay utilizando LiveData.

# Dependencia

Para incorporar la funcionalidad deseada, agrega la siguiente dependencia a tu archivo build.gradle:

implementation "androidx.lifecycle:lifecycle-livedata-core-ktx:2.6.2"

Este paso es esencial para utilizar las características más recientes y eficientes de LiveData en tu aplicación Android.

Propiedades Mutables

Propiedades LiveData y Acceso a Valores

Para utilizar propiedades LiveData, es necesario emplear una clase genérica junto con el tipo de dato que se desea almacenar.

var secretWordDisplay = MutableLiveData<String>()

var lives = MutableLiveData<Int>(8)

Acceder a los valores de estas propiedades requiere el uso de métodos específicos:

secretWordDisplay.value = generateSecretWordDisplay()

En el caso de propiedades que podrían contener valores nulos, como lives, se utiliza el operador Elvis para manejar nulos:

lives.value = lives.value?.minus(1)

fun lost() = lives.value ?: 0 <= 0

Observadores

Se pueden definir observadores para realizar acciones específicas cuando los datos cambian. Aquí hay un ejemplo de cómo crear un observador para un tipo de dato específico:

val nameObserver = Observer<String> { newName ->

// Actualizar la interfaz de usuario, en este caso, un TextView.

nameTextView.text = newName

}

Este observador, en este caso, llamado nameObserver, reaccionará a los cambios en el LiveData de tipo String. Puedes personalizar las acciones dentro del bloque lambda para actualizar la interfaz de usuario según tus necesidades.

Capitulo 7

UD07. Persistencia de Datos

Resultados de avaliación

**RA1.** Aplica tecnoloxías de desenvolvemento para dispositivos móbiles, e avalía as súas características e as súas capacidades.

**RA2.** Desenvolve aplicacións para dispositivos móbiles, para o que analiza e emprega as tecnoloxías e as librarías específicas.

Criterios de avaliación

* CA1.8 Realizáronse modificacións sobre aplicacións existentes.
* CA1.9 Utilizáronse emuladores para comprobar o funcionamento das aplicacións.
* CA2.1 Xerouse a estrutura de clases necesaria para a aplicación.
* CA2.4 Utilizáronse as clases necesarias para a conexión e a comunicación con dispositivos sen fíos.
* CA2.5 Utilizáronse as clases necesarias para o intercambio de mensaxes de texto e multimedia.
* CA2.8 Realizáronse probas de interacción entre o usuario e a aplicación para mellorar as aplicacións desenvolvidas a partir de emuladores.
* CA2.9 Empaquetáronse e despregáronse as aplicacións desenvolvidas en dispositivos móbiles reais.
* CA2.10 Documentáronse os procesos necesarios para o desenvolvemento das aplicacións

BC1. Análise de tecnoloxías para desenvolvemento de aplicacións en dispositivos móbiles.

* Contidos
* Contornos integrados de traballo.
* Módulos para o desenvolvemento de aplicacións móbiles.
* Emuladores.
* Estrutura dunha aplicación para dispositivo móbil.
* Modificación de aplicacións existentes.
* Uso do contorno de execución do administrador de aplicacións.
* Ferramentas e fases de construción.
* Técnicas de animación e son.
* Comunicacións: clases asociadas. Tipos de conexións.
* Xestión da comunicación sen fíos.
* Envío e recepción de mensaxes de texto: seguridade e permisos.
* Envío e recepción de mensaxaría multimedia: sincronización de contido; seguridade e permisos.

# Compose

En el contexto de Android, “Compose” generalmente se refiere a Jetpack Compose, que es un moderno kit de herramientas de UI (interfaz de usuario) declarativo y totalmente nativo para el desarrollo de aplicaciones Android. Jetpack Compose permite a los desarrolladores construir interfaces de usuario de manera más sencilla y más eficiente mediante la creación de interfaces de usuario mediante código Kotlin.

Para poder realizar un proyecto con Compose debemos seleccionar la plantilla “Empty View”.

# Previsualización

En el código proporcionado, la opción de “Previsualización” se refiere a la capacidad de Jetpack Compose para proporcionar una vista previa visual en tiempo de diseño de cómo se verá un componente en la interfaz de usuario antes de que se ejecute la aplicación. La función @Preview se utiliza para definir vistas previas en Compose.

En el código que compartiste:

@Preview(showBackground = true)

@Composable

fun GreetingPreview() {

Prueba\_composeTheme {

Greeting("Android")

}

}

Aquí, @Preview se utiliza para crear una vista previa de la función Greeting. La función GreetingPreview se compone de la función Greeting y se le proporciona un nombre ("Android") como argumento para simular cómo se vería la interfaz de usuario cuando se utiliza la función Greeting con ese argumento específico.

La opción showBackground = true indica que se debe mostrar un fondo en la vista previa, lo que puede ser útil para visualizar cómo se integrará el componente en el diseño general de la interfaz de usuario.

# Surface

En el contexto de Jetpack Compose, una Surface es un componente que actúa como un contenedor visual y define un área rectangular en la interfaz de usuario. La función principal de la Surface es proporcionar un fondo para otros componentes y establecer características visuales para ese área específica.

Algunas características clave de Surface incluyen:

1. **Definición de Fondo:** Puedes especificar un color de fondo para la Surface, que afectará a la apariencia visual de cualquier componente contenido en ella.
2. **Gestión de Propiedades Visuales:** La Surface puede gestionar propiedades visuales como sombras, bordes y más, lo que permite personalizar la apariencia del área que cubre.
3. **Contenedor para Otros Componentes:** Puedes anidar otros componentes dentro de una Surface, lo que permite componer interfaces de usuario más complejas y estructuradas.

En el contexto de nuestro código:

@Composable

fun Greeting(name: String, modifier: Modifier = Modifier) {

// La Surface toma un color, en este caso, el color primary del tema.

Surface(color = MaterialTheme.colorScheme.primary) {

// Los componentes anidados dentro de Surface se dibujarán sobre ese color de fondo.

Text(

text = "Hello $name!",

modifier = modifier

)

}

}

Aquí, la Surface se utiliza para definir el color de fondo (en este caso, el color primario del tema) sobre el cual se dibujará el componente Text.

# Modificadores

En Compose, la mayoría de los elementos de la interfaz de usuario, como Surface y Text, incorporan un parámetro opcional denominado modifier. Estos modificadores permiten especificar cómo debe presentarse o comportarse un elemento en el diseño general.

Text(

text = "Hello $name!",

modifier = modifier.padding(Dp(15f))

)

En el ejemplo anterior, se aplica un modificador para establecer un relleno (padding) alrededor del componente Text, proporcionando así un espacio adicional alrededor del texto.

[Consulta la lista completa de modificadores](https://developer.android.com/jetpack/compose/modifiers-list?hl=es-419) para explorar las diversas opciones disponibles para personalizar la apariencia y el comportamiento de los elementos de la interfaz de usuario en Jetpack Compose.

# Componibles

Con el objetivo de promover la **reutilización de código**, se propone la creación de un elemento **componible** llamado MyApp que encapsule la funcionalidad de mostrar un saludo. Como una práctica recomendada, se sugiere incluir un parámetro de modificador en la función, asignándole un modificador vacío de forma predeterminada.

@Composable

private fun MyApp(modifier: Modifier = Modifier) {

Surface(

modifier = modifier,

color = MaterialTheme.colorScheme.background

) {

Greeting("Android")

}

}

Esta implementación posibilita la simplificación de la devolución de llamada onCreate y la vista previa, ya que ahora es posible **reutilizar** el elemento componible MyApp, evitando la duplicación de código.

En la clase MainActivity, el uso de MyApp se integra de la siguiente manera:

class MainActivity : ComponentActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContent {

// En lugar de un archivo XML, se invoca a una función de Compose

Prueba\_composeTheme {

// Elementos componibles

MyApp(modifier = Modifier.fillMaxSize())

}

}

}

}

Asimismo, en la vista previa, se puede observar la reutilización de MyApp de la siguiente manera:

@Preview(showBackground = true)

@Composable

fun GreetingPreview() {

Prueba\_composeTheme {

// Elementos componibles

MyApp()

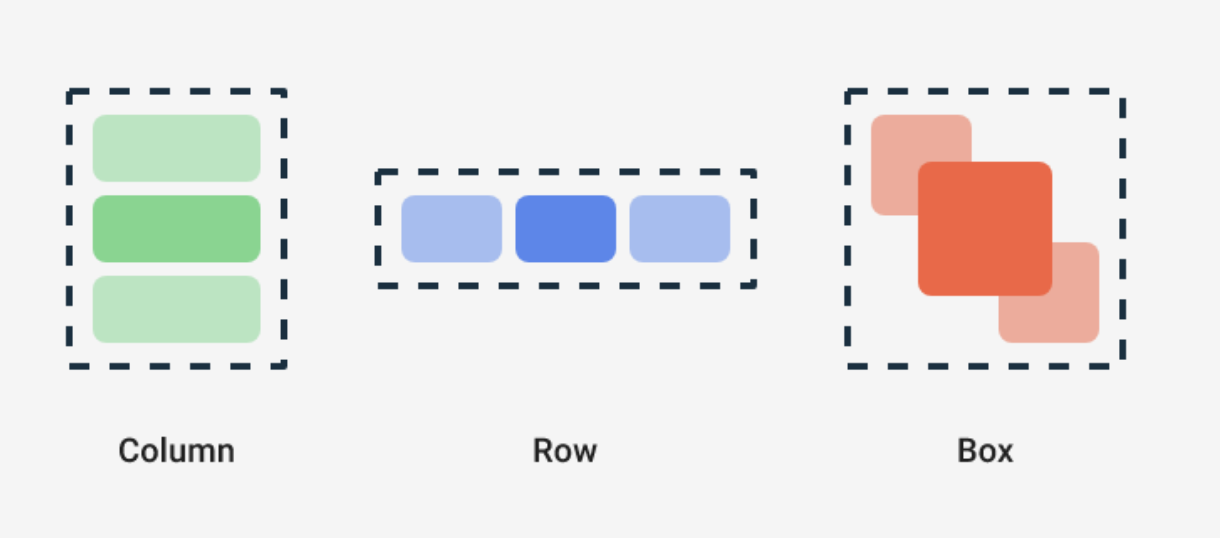
}

}

Esta estructura fomenta la coherencia y facilita la gestión de componentes componibles en la aplicación, mejorando la legibilidad y la mantenibilidad del código.

# Elementos Básicos

Los componentes fundamentales en Compose son **Column**, **Row** y **Box**, representando estructuras de diseño estándar.

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud07/compose/5.elementos_basicos/index.html#R-image-f5f0ea6876641aadf66fca4d2ecb0aab)

Pueden combinarse para lograr disposiciones más complejas y flexibles. En el siguiente código, ilustramos cómo crear una columna dentro de la función Greeting:

@Composable

fun Greeting(name: String, modifier: Modifier = Modifier) {

Surface(color = MaterialTheme.colorScheme.primary) {

Column(modifier = Modifier.padding(Dp(24f))) {

Text(text = "Hello,")

Text(text = name)

}

}

}

En este ejemplo, hemos encapsulado dos elementos de texto (Text) dentro de una columna (Column). La columna se ha configurado con un modificador para agregar un relleno alrededor de los elementos. La superficie (Surface) establece el color de fondo de la composición.

Capitulo 2

Room

En el contexto de desarrollo de aplicaciones Android, “Room” se refiere a una biblioteca de persistencia que proporciona una capa de abstracción sobre SQLite, que es una base de datos relacional incorporada en Android. Room simplifica el manejo de la base de datos y ofrece una forma más robusta y eficiente de realizar operaciones de base de datos en comparación con el uso directo de SQLite.

Las principales características de Room incluyen:

1. **Entity:** Representa una tabla en la base de datos. Cada instancia de la entidad representa una fila en esa tabla.
2. **DAO (Data Access Object):** Define métodos que acceden a la base de datos. Estos métodos pueden incluir consultas SQL personalizadas.
3. **Database:** Es una clase que sirve como punto de acceso principal para la base de datos. Se anota con la anotación @Database y debe ser una extensión de la clase RoomDatabase. La clase Database generalmente incluye una lista de entidades y proporciona un método abstracto que devuelve la instancia de la interfaz DAO asociada.

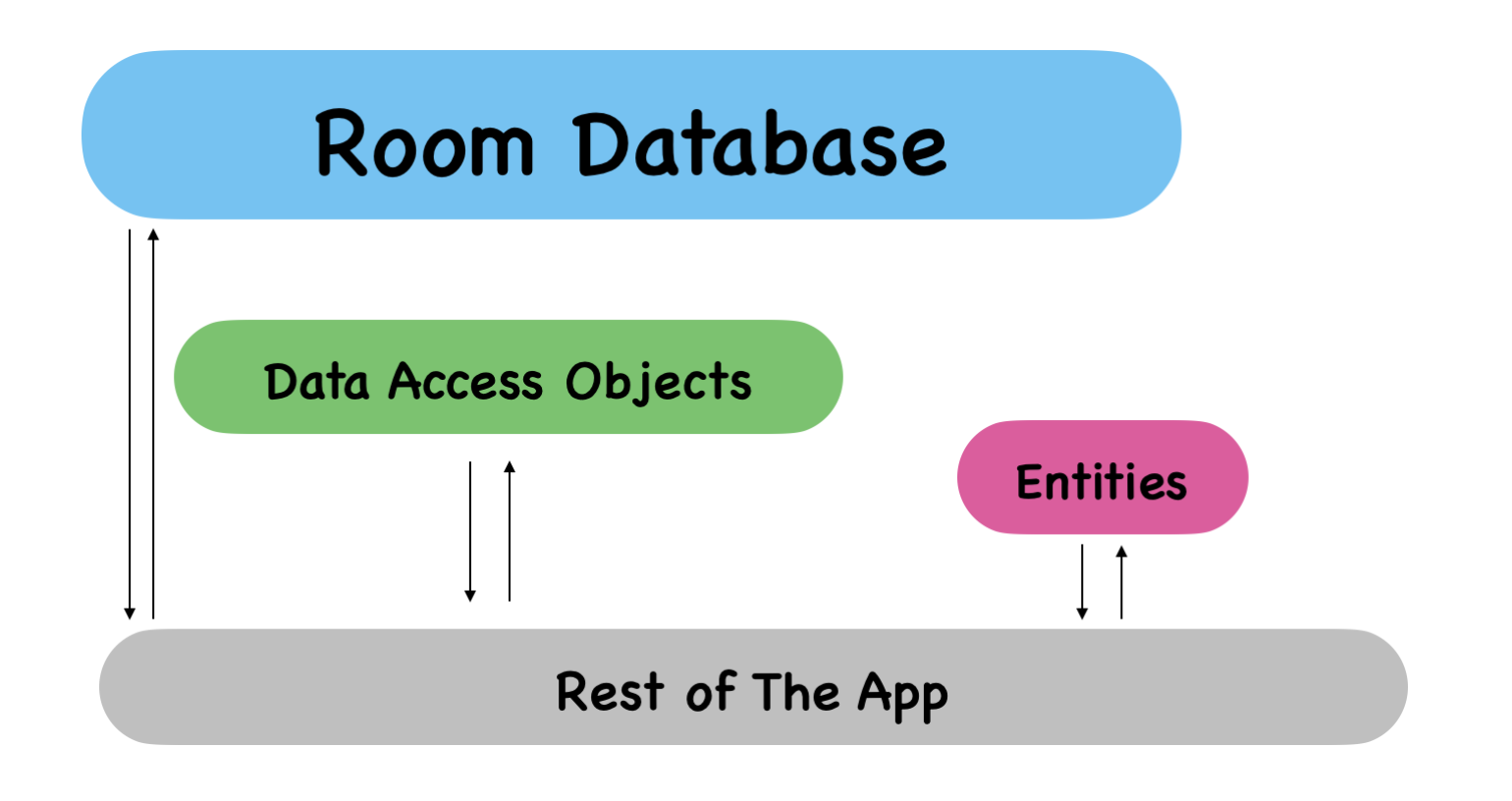
Room simplifica muchas tareas comunes en el manejo de bases de datos, como la creación de tablas, la ejecución de consultas, la gestión de transacciones y la actualización de esquemas de base de datos. También proporciona una capa de abstracción sólida que facilita el cambio del proveedor de la base de datos subyacente (por ejemplo, de SQLite a otro).

Para utilizar Room en un proyecto Android, debes agregar las dependencias necesarias en tu archivo build.gradle y luego crear las entidades, DAO y la clase de base de datos según las necesidades de tu aplicación.

# Componentes principales

Kotlin ofrece una forma sencilla de gestionar datos mediante la introducción de clases de datos. Puedes acceder y modificar estos datos fácilmente utilizando llamadas a funciones. Sin embargo, en el contexto de las bases de datos, la manipulación de datos requiere la interacción con tablas y consultas. Room, una biblioteca de persistencia en Android, simplifica estos flujos de trabajo, y sus principales componentes son cruciales para ello.

Los tres componentes principales de Room son los siguientes:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud07/room/1.componentes/index.html#R-image-25a0b730666b00281531ccb097cc05d6)

* **Entidades:** Estas representan las tablas de la base de datos de tu aplicación. Se utilizan para actualizar los datos almacenados en filas dentro de las tablas y para crear nuevas filas que se insertarán.
* **Objetos de Acceso a Datos (DAO):** Los DAO proporcionan métodos que tu aplicación utiliza para recuperar, actualizar, insertar y eliminar datos en la base de datos. Son una interfaz entre tu código y la lógica de persistencia.
* **Clase de Base de Datos:** Esta clase contiene la base de datos y sirve como el principal punto de acceso para la conexión subyacente a la base de datos de la aplicación. Además, la clase de base de datos proporciona instancias de los DAO asociados, facilitando la ejecución de operaciones en la base de datos de manera organizada y eficiente.

# Dependencia

En el bloque de configuración de Gradle de tu proyecto, debes incluir las bibliotecas necesarias para los componentes de Room. Aquí se muestra cómo integrar las bibliotecas de Room en tus archivos de Gradle:

Para el bloque de configuración de Gradle del proyecto:

plugins {

id("com.google.devtools.ksp") version "1.8.10-1.0.9" apply false

}

// Otras configuraciones de tu proyecto...

dependencies {

// Otras dependencias de tu proyecto...

}

Para el bloque de configuración de Gradle de la aplicación:

plugins {

id("com.google.devtools.ksp")

}

// Otras configuraciones de tu aplicación...

dependencies {

// Room libraries

implementation("androidx.room:room-runtime:2.6.0")

ksp("androidx.room:room-compiler:2.6.0")

implementation("androidx.room:room-ktx:2.6.0")

implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.6.2")

// Otras dependencias de tu aplicación...

}

Estas configuraciones aseguran que las bibliotecas de Room necesarias estén incluidas correctamente en tu proyecto o aplicación, según la naturaleza del bloque de configuración. Asegúrate de ajustar las versiones de las bibliotecas según tus requisitos específicos.

# Entity

La clase Entity sirve para definir una tabla en la base de datos, donde cada instancia de esta clase representa una fila en dicha tabla. Además, cuenta con anotaciones que le indican a Room cómo debe presentar y manipular la información en la base de datos. En el contexto de tu aplicación, esta entidad almacenará datos relacionados con los elementos del inventario, tales como el nombre, el precio y la cantidad disponible.

A continuación, se describen los pasos para implementar esta entidad llamada Item:

1. Crea un paquete llamado data dentro del paquete base de tu aplicación:

[](https://manuais.pages.iessanclemente.net/plantillas/dam/pmdm/ud07/room/3.entity/index.html#R-image-db1984a72a99377ec4a0653684bf6958)

1. Dentro del paquete data, crea una nueva clase Kotlin llamada Item, la cual representará una entidad en tu base de datos.
2. Actualiza la definición de la clase Item con el siguiente código, declarando id como Int, itemName como String, itemPrice como Double, y quantityInStock como Int, asignando valores predeterminados según sea necesario:
3. class Item {
4. val id: Int = 0
5. val itemName: String = ""
6. val itemPrice: Double = 0.0
7. val quantityInStock: Int = 0

}

1. Convierte la clase Item en una clase de datos añadiendo la palabra clave data al principio de su definición:
2. data class Item(
3. val id: Int = 0,
4. val itemName: String,
5. val itemPrice: Double,
6. val quantityInStock: Int

)

1. Anota la clase de datos con @Entity sobre la declaración de la clase. Utiliza el argumento tableName para designar “item” como el nombre de la tabla en SQLite:
2. @Entity(tableName = "item")
3. data class Item(
4. val id: Int = 0,
5. val itemName: String,
6. val itemPrice: Double,
7. val quantityInStock: Int

)

1. Identifica id como la clave primaria anotándola con @PrimaryKey. Configura el parámetro autoGenerate en true para permitir que Room genere automáticamente el ID de cada entidad:
2. @Entity(tableName = "item")
3. data class Item(
4. @PrimaryKey(autoGenerate = true)
5. val id: Int = 0,
6. val itemName: String,
7. val itemPrice: Double,
8. val quantityInStock: Int

)

1. Anota las propiedades restantes con @ColumnInfo. Esta anotación se utiliza para personalizar el nombre de la columna asociada con cada campo:
2. @Entity(tableName = "item")
3. data class Item(
4. @PrimaryKey(autoGenerate = true)
5. val id: Int = 0,
6. @ColumnInfo(name = "name")
7. val itemName: String,
8. @ColumnInfo(name = "price")
9. val itemPrice: Double,
10. @ColumnInfo(name = "quantity")
11. val quantityInStock: Int

)

Estos pasos establecen la estructura de la entidad Item para tu base de datos Room, definiendo las propiedades, la clave primaria y las personalizaciones de las columnas según sea necesario.

# DAO

Para realizar operaciones comunes en la base de datos, Room ofrece convenientes anotaciones como @Insert, @Delete y @Update. Para casos más específicos, se utiliza la anotación @Query, que permite escribir consultas compatibles con SQLite.

Ahora, procede a implementar el Data Access Object (DAO) en tu aplicación. Sigue estos pasos:

En el paquete data, crea la clase Kotlin ItemDao.kt. Transforma la definición de la clase en una interfaz y anótala con @Dao para indicar que es un DAO.

@Dao

interface ItemDao {

}

Introduce las funciones que realizarán las operaciones en la base de datos. Usa la anotación @Query para las consultas personalizadas. Aquí tienes un ejemplo:

@Dao

interface ItemDao {

@Query("SELECT \* from item ORDER BY name ASC")

fun getItems(): Flow<List<Item>>

@Query("SELECT \* from item WHERE id = :id")

fun getItem(id: Int): Flow<Item>

// Especifica la estrategia de conflicto como IGNORE. Cuando el usuario intenta agregar un

// Item existente a la base de datos, Room ignora el conflicto.

@Insert(onConflict = OnConflictStrategy.IGNORE)

suspend fun insert(item: Item)

@Update

suspend fun update(item: Item)

@Delete

suspend fun delete(item: Item)

}

En estas funciones:

* getItems: Recupera todos los elementos ordenados por nombre en forma de flujo (Flow).
* getItem: Recupera un elemento específico por su ID en forma de flujo (Flow).
* insert: Inserta un nuevo elemento, ignorando conflictos si el elemento ya existe.
* update: Actualiza la información de un elemento existente.
* delete: Elimina un elemento de la base de datos.

La utilización de suspend indica que estas funciones pueden pausar su ejecución y reanudarse más tarde, lo cual es esencial para operaciones en la base de datos que pueden bloquear el hilo principal.

Instancia Base de Datos

En el paquete data, crea la clase Kotlin InventoryDatabase.kt. Luego, en el archivo InventoryDatabase.kt, convierte la clase InventoryDatabase en una clase abstracta que extienda RoomDatabase y anótala con @Database. A pesar de la presencia de un error de parámetros faltantes, se corregirá en el siguiente paso:

@Database(entities = [Item::class], version = 1, exportSchema = false)

abstract class InventoryDatabase : RoomDatabase() {

abstract fun itemDao(): ItemDao

companion object {

@Volatile

private var instance: InventoryDatabase? = null

fun getDatabase(context: Context): InventoryDatabase {

// Aún no se ha implementado

}

}

}

La anotación @Database requiere varios argumentos para que Room pueda compilar la base de datos:

* Especifica Item como la única clase con la lista de entidades.
* Establece version en 1. Debes incrementar el número de versión cada vez que cambies el esquema de la tabla de la base de datos.
* Establece exportSchema como false para que no se conserven copias de seguridad del historial de versiones de esquemas.

Dentro del cuerpo de la clase, se declara una función abstracta que proporciona el ItemDao, permitiendo que la base de datos conozca el DAO asociado con la entidad Item.

El objeto companion se utiliza para acceder a métodos para crear u obtener la base de datos, utilizando el nombre de la clase como calificador. Se define una variable anulable privada instance para la base de datos dentro del objeto companion, inicializándola en null. La palabra clave volatile se utiliza para garantizar que la instancia de la base de datos sea siempre visible para otros subprocesos de manera consistente.

Debajo de instance, dentro del objeto companion, se define un método getDatabase() con un parámetro Context necesario para el compilador de bases de datos, mostrando un tipo InventoryDatabase. En este punto, el método aún no está implementado.

Codelabs

* [Aspectos Básicos Jetpack Compose](https://developer.android.com/codelabs/jetpack-compose-basics?hl=es-419#5)
* [Room y Compose](https://developer.android.com/codelabs/basic-android-kotlin-compose-persisting-data-room?hl=es-419&continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-6-pathway-2%3Fhl%3Des-419%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-persisting-data-room#0)
* [Conservar datos con Room](https://developer.android.com/codelabs/basic-android-kotlin-training-persisting-data-room?hl=es-419#0)

# GIT

[Apuntes sobre GIT](https://manuais.pages.iessanclemente.net/apuntes/2.programacion/git/index.html)