

## Desarrollo de la interfaz de usuario con Jetpack Compose

Una función componible es cualquier función marcada con la `@Compose` anotación. En los capítulos anteriores, creaste y usaste funciones componibles para realizar tareas dentro de los ejercicios. Algunas de las...Los componentes componibles utilizados fueron componentes integrados como `Text` y `Button` y grupos de diseño integrados de `Row` y `Column`. Luego, creaste tus propias funciones componibles usando estos bloques de construcción.

Este capítulo ofrece una visión más detallada de las funciones componibles básicas y los grupos de diseño en Jetpack Compose. Demuestra cómo usarlos para crear la interfaz de usuario y responder a los cambios de estado mediante su recomposición.

Al final de este capítulo, podrá utilizar todas las funciones componibles principales para crear elementos de interfaz de usuario y habrá aprendido a diseñar estos elementos utilizando los principales grupos de diseño componibles.

En este capítulo cubriremos los siguientes temas:

- Transición de diseños XML a Jetpack Compose
- Funciones componibles esenciales
- Grupos de diseño de Jetpack Compose

Para comenzar, aprenderá sobre el proceso heredado de usar XML para crear diseños y comparará su enfoque imperativo con el proceso declarativo de usar Jetpack Compose.

### Requisitos técnicos

El código completo de todos los ejercicios y la actividad de este capítulo está disponible en GitHub en <https://packt.link/9GoAL>.

## Transición de diseños XML a Jetpack Compose

Antes de que Jetpack Compose se convirtiera en el marco de interfaz de usuario establecido, las aplicaciones de Android usaban XML para la interfaz de usuario. EstasArchivos XML contenidosUn esquema y un elemento raíz con elementos XML incrustados para el contenido. El XML debía cargarse.Al iniciarse la actividad, mediante un proceso denominado **inflación**, el XML se transformó en una jerarquía de vistas de Android. Era necesario recuperar las vistas antes de poder interactuar con ellas para configurar datos y realizar cualquier operación.

Es importante conocer los conceptos básicos de cómo funciona esto, ya que muchas aplicaciones existentes se crearon utilizando solo este patrón, y la mayoría de las aplicaciones establecidas de Google Play tendrán algún código XML heredado después de adoptar Compose.

### Ejercicio 3.01 – Creación de una aplicación de contador con vistas heredadas

Para elPrimer ejercicio: crearás una aplicación de contador sencilla queutiliza vistas usando una nueva plantilla de proyecto:

1. Abre Android Studio y selecciona "Nuevo proyecto" en la pantalla de bienvenida de Android. Selecciona "Actividad de vistas vacías" y llámala "Counter Views".
2. Agregue las siguientes cadenas al `value/strings.xml` archivo:

```
<string name="zero">0</string>
<string name="plus">+</string>
<string name="minus">-</string>
<string name="counter_text">Counter</string>
```

3. Reemplaza los TextView elementos XML en el `activity_main.xml` archivo, que es dentro de la `res/layout` carpeta, con estos nuevos TextView elementos, que muestran una etiqueta y el valor del contador:

```
<TextView  
    android:id="@+id/counter_text"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/counter_text"  
    android:paddingTop="10dp"  
    android:textSize="44sp"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    app:layout_constraintBottom_toTopOf="@+id/counter_value" />  
  
<TextView  
    android:id="@+id/counter_value"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/zero"  
    android:textSize="54sp"  
    android:textStyle="bold"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    app:layout_constraintBottom_toBottomOf="parent" />
```

4. A continuación, agreguelos Button elementos que aumentarán y disminuirán el valor del contador debajo de los TextView elementos:

```
<Button  
    android:id="@+id/plus"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/plus"  
    android:textSize="40sp"  
    android:textStyle="bold"  
    app:layout_constraintEnd_toStartOf="@+id/minus"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/counter_value"  
    app:layout_constraintBottom_toBottomOf="parent" />  
  
<Button  
    android:id="@+id/minus"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/minus"  
    android:textSize="40sp"  
    android:textStyle="bold"  
    app:layout_constraintStart_toEndOf="@+id/plus"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/counter_value"  
    app:layout_constraintBottom_toBottomOf="parent" />
```

5. Correrla aplicación en el emulador y tú Verá la siguiente pantalla:



Figura 3.1 – Una aplicación de contador simple

El diseño es bastante simple. Hay son cuatro vistas, contenidas en un grupo de diseño principal llamado `ConstraintLayout`. Este es el grupo de diseño XML más común (o `ViewGroup`, como se les llama en la jerarquía de vistas XML). Todas las vistas incrustadas tienen un `id` identificador con el que se pueden recuperar en la actividad, y utilizan los valores de restricción superior, inferior, inicial y para limitarse a su padre o a las otras vistas con su `id` valor de identificador, se pueden añadir directrices y barreras para posicionar las vistas con mayor precisión. Con estas restricciones se pueden lograr diseños complejos, pero el XML... Entonces, "required" se vuelve extenso y menos legible.

6. Regresa `MainActivity` y agrega este código al final de `onCreate()`:

```
var counter = 0
val mainView = findViewById<ConstraintLayout>(R.id.main)
val counterValue = mainView.findViewById<TextView>(
    R.id.counter_value
)
val plusButton = mainView.findViewById<Button>(
    R.id.plus
)
val minusButton = mainView.findViewById<Button>(
    R.id.minus
)
plusButton.setOnClickListener {
    counter++
    counterValue.text = counter.toString()
}
minusButton.setOnClickListener {
    if (counter > 0) {
        counter--
        counterValue.text = counter.toString()
    }
}
```

Se crea una `counter` variable que, al tener un `var` tipo, es mutable y actualizable. Todas las vistas requeridas se recuperan de la jerarquía de vistas con `findViewById`, y `OnClickListener` se añade una devolución de llamada a cada `Button` vista con la `setOnClickListener` función que se invocará al `Button` presionar el objeto. Esto actualiza el `counter` entero y muestra el resultado en la `counterValue` vista.

En este sencillo ejemplo, el enfoque funciona bien para crear funcionalidades básicas. Sin embargo, presenta algunas desventajas. La interfaz de usuario (IU) aún se divide entre el código fuente y el XML, en lugar de generarse en código. También puede requerir alternar entre códigos y el editor XML, lo que puede ralentizar el desarrollo. La principal desventaja en comparación con Compose es la naturaleza imperativa de la creación de la interfaz de usuario. Cada vista que requiere actualización debe recuperarse y luego configurarse. Cada cambio en la interfaz de usuario requiere instrucciones paso a paso sobre cómo gestionar los cambios de estado. Es un enfoque manual que puede generar mucho código repetitivo y lógica integrada en las vistas, lo cual puede ser propenso a errores.

## Ejercicio 3.02 – Creación de una aplicación Android con Compose

Vamos a examinar la creación de la misma aplicación de ejemplo implementada en Componer:

1. Abre Android Studio y selecciona "Nuevo proyecto" en la pantalla de bienvenida de Android. Selecciona "Actividad vacía" y llámala "Counter Compose".
2. Agregue las mismas cadenas requeridas al `value/strings.xml` archivo:

```
<string name="zero">0</string>
<string name="plus">+</string>
<string name="minus">-</string>
<string name="counter_text">Counter</string>
```

3. Crea un componente componible llamado `MainScreen` a continuación la `MainActivity` clase:

```

@Composable
fun MainScreen(modifier: Modifier) {
    var counter by remember { mutableStateOf(0) }
    Column(
        verticalArrangement = Arrangement.SpaceEvenly,
        horizontalAlignment = Alignment.CenterHorizontal
        modifier = modifier.fillMaxHeight()
    ) {
        Text(
            text = stringResource(id = R.string.counter_t
            fontSize = 44.sp,
            modifier = Modifier.align(Alignment.CenterHor
        )
        Text(
            text = counter.toString(),
            fontSize = 54.sp,
            fontWeight = FontWeight.Bold,
        )
        Row(
            horizontalArrangement = Arrangement.SpaceEven
            modifier = Modifier.fillMaxWidth()
        ) {
            Button(
                onClick = { counter++ }
            )
            Text(
                text = stringResource(id = R.string.p
                fontSize = 44.sp,
                fontWeight = FontWeight.Bold,
                modifier = Modifier.padding(horizontal
            )
        }
        Button(
            onClick = {
                if (counter > 0) {
                    counter--
                }
            }
        )
        Text(
            text = stringResource(id = R.string.m
            fontSize = 44.sp,
            modifier = Modifier.padding(horizontal
            fontWeight = FontWeight.Bold,
        )
    }
}
}

```

Es recomendable crear elementos componibles separados al crear su interfaz de usuario, siempre que sea posible, comenzando con el elemento componible más pequeño. Funciones. Esto se denomina **enfoque ascendente**. Crear pequeños componentes componibles fomenta la reutilización y, al minimizar... El estado dentro de ellos (ya que el estado puede transmitirse como argumentos) hace que el código sea más robusto. La creación de componentes componibles más pequeños también proporciona información explicativa. uso a través del nombre, algo así como una refactorización general para separar grandes troncos de código en funciones más pequeñas.

De forma similar a la `Counter Views` aplicación, se crea un contador para almacenar el estado. La diferencia en el ejemplo de Compose es que no se establece como un entero, sino como un contenedor de estado con `var counter by remember { mutableStateOf(0) }`. Al usar la `remember` función, el valor se conservará al recomponer la interfaz de usuario. En el `Counter Views` ejemplo, después de aumentar o disminuir el valor del contador, era necesario establecerlo explícitamente con el siguiente código, que ya no es necesario:

```
counterValue.text = counter.toString()
```

Otra ventaja es que la estructura y el comportamiento de la interfaz de usuario son claros. No es necesario alternar entre el código y un archivo de diseño XML para comprenderla. Consiste en dos `Text` elementos componibles dentro de otro `Row` elemento componible, que a su vez está dentro de otro `Column` elemento componible. El contador se incrementa y decremente haciendo clic en los botones más (`counter++`) y menos (`counter--`). Estas acciones activan una recomposición, ya que el valor `counter` de solo lectura `Text` depende del estado.

4. Para configurar el contenido de la actividad a mostrar `MainScreen`, agregue una llamada al componible dentro de `onCreate()`:

```
Scaffold(  
    modifier = Modifier.fillMaxSize()  
) { innerPadding ->  
    MainScreen(  
        Modifier.padding(innerPadding)  
    )  
}
```

Pasar `innerPadding` a nivel inferior `MainScreen` garantiza que el elemento componible utilice el máximo espacio disponible, considerando elementos de la interfaz de usuario del sistema, como la barra de estado. `Scaffold` es un elemento componible de nivel superior que permite posicionar contenido, como `TopAppBar` la barra de navegación inferior, para proporcionar un diseño estructurado a la aplicación. Aprenderá sobre esto en [el Capítulo 4, Creación de navegación de aplicaciones](#).

5. Como paso final, agregue una vista previa componible al archivo agregando una función componible con la `@Preview` anotación y agregando el `MainScreen` elemento componible dentro de ella:

```
@Preview  
@Composable  
fun MainScreenPreview() {  
    MainScreen(modifier = Modifier.padding(20.dp))  
}
```

Las `@Preview` funciones anotadas permiten ver el elemento componible en el editor con cambios en vivo una vez que se ha creado la aplicación.

El precedente El ejemplo es declarativo porque describe cómo debería verse la interfaz de usuario según el estado. El enfoque se basa en reaccionar a los cambios de estado, en lugar de...que el enfoque imperativo, que requiere agregar pasos para actualizar el estado.

En la siguiente sección, explorará la variedad de elementos componibles integrados que están disponibles para crear interfaces de usuario enriquecidas y atractivas.

## Funciones componibles esenciales

El kit de herramientas de diseño de interfaz de usuario de Android utiliza una combinación de elementos básicos y fundamentales componibles y aquellos que...Incorporar Material Design. Existe una gran cantidad de funciones componibles para crear todo tipo de elementos Material UI, denominados **componentes** en el lenguaje Material Design . En este...En esta sección, nos centraremos en los componentes fundamentales de Compose, cruciales para la visualización, la gestión de la interacción del usuario y la recopilación de sus entradas. Para ver la lista completa de componentes de Material, visite <https://packt.link/h0mK0>.

Puede encontrar detalles completos del sistema Google Material Design 3 aquí: <https://m3.material.io/>.

Si alguna vez necesitas crear un componible desde el nivel más bajo sin temática Material, los componibles para hacerlo se pueden encontrar en este paquete: <https://packt.link/kU10z>.

### Texto

Tú tienes Ya se ha usado el `Text` componente componible en muchos ejemplos. En su forma más simple, solo se necesita establecer el `text` parámetro:

```
Text(text = "Hello world!")
```

Luego muestra el texto en el estilo predeterminado:

Hello world!

Figura 3.2 – Un texto básico componible

Sin embargo, el `Text` elemento componible incluye muchas otras funciones. Si selecciona la fuente del elemento `Text` componible (posicionando el cursor en el nombre de la función y luego presionando *comando-clic* en una Mac o *Ctrl-clic* en Windows o Linux), verá todas sus propiedades:

```
// This code is part of the Android SDK and is licensed under
// Copyright (C) 2024 Google Inc.
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified,
    fontStyle: FontStyle? = null,
    fontWeight: FontWeight? = null,
    fontFamily: FontFamily? = null,
    letterSpacing: TextUnit = TextUnit.Unspecified,
    textDecoration: TextDecoration? = null,
    textAlign: TextAlign? = null,
    lineHeight: TextUnit = TextUnit.Unspecified,
    overflow: TextOverflow = TextOverflow.Clip,
    softWrap: Boolean = true,
    maxLines: Int = Int.MAX_VALUE,
    minLines: Int = 1,
    onTextLayout: ((TextLayoutResult) -> Unit)? = null,
    style: TextStyle = LocalTextStyle.current
)
```

Vamos a ver el más comúnmente utilizado de estos parámetros:

- `text`: El texto que se mostrará.
- `modifier`: Se utiliza para personalizar el diseño y el comportamiento de los elementos componibles. En cuanto al diseño, permite configurar el relleno, el tamaño, la alineación, el fondo, el borde y otras opciones de visualización. En cuanto al comportamiento, permite hacer clic en un elemento componible y configurar otras interacciones, como arrastrarlo o enfocarlo.
- `color`: Permite configurar el color del texto.
- `fontSize`: Establece el tamaño de fuente en píxeles escalables.
- `fontStyle`: Establece el estilo de fuente en `Normal` o `Italic`.
- `fontWeight`: Establece el grosor o peso de la fuente, que normalmente se utiliza para establecer el texto `Bold`, pero los valores que se pueden configurar varían desde `Thin` el más claro hasta `Black` el más pesado.
- `textAlign`: Establece cómo debe alinearse el texto horizontalmente.
- `overflow`: Especifica cómo la aplicación maneja el texto desbordado, utilizando `Clip` o `Ellipsis`.
- `style`: Permite configurar un estilo que puede establecer múltiples propiedades (y reutilizarse para otros elementos componibles) `Text` en lugar de configurarlas individualmente.

(Puedes mirar (Los otros parámetros se describen con más detalle en la documentación de Kotlin, Kdoc, encima del nombre de la función). El componente de bajo nivel correspondiente `Text` sin estilo Material tiene menos parámetros y se llama `BasicText`.

## Botón

Un `Button` componible activa una acción cuando el usuario interactúa con él. Al usar XML para la interfaz de usuario, el botón es un elemento XML:

```
<Button
    android:id="@+id/buttonSubmit"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Save" />
```

Luego debes agregar un `clickListener` objeto en el botón para crear la interacción:

```
val button: Button = findViewById(R.id.buttonSubmit)
button.setOnClickListener {
    // Button click handled here
}
```

Puedes ver el imperativoLa naturaleza del uso de vistas de Android para crear la interfaz de usuario (IU) es fundamental. Debe actualizar la IU manualmente. En

Jetpack Compose, `Button` se puede hacer clic en el elemento componible simplemente añadiendo el `onClick` argumento y ajustando el contenido:

```
Button(onClick = { /* Handle click */ }) {  
    Text("Save")  
}
```

La siguiente figura muestra el botón que se muestra como resultado del código anterior:



Figura 3.3 – Botón con estilo de material

El `Button` tipo componible es el básico, y existen otros botones que `Button` los usan como base y añaden contornos, elevación y rellenos. Consulta la documentación aquí para obtener más información: <https://packt.link/cgeN4>.

### Icono

`Icon` pantalla un ícono, que es una imagen pequeña que se utiliza para dar significado contextual a los elementos de la interfaz de usuario y, opcionalmente, agregar comportamiento.

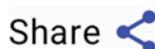


Figura 3.4 – Un ícono componible para compartir de color azul

El ícono de la figura anterior se creó con el siguiente código:

```
Row {  
    Text(  
        text = "Share",  
        Modifier.padding(end = 4.dp)  
    )  
    Icon(  
        modifier = Modifier.clickable { /* Handle click */ }  
        imageVector = Icons.Default.Share,  
        tint = Color.Blue,  
        contentDescription = "Share"  
    )  
}
```

En este ejemplo, `Text` se muestra con un `Share` ícono que establece el `imageVector` parámetro a utilizar un recurso vectorial integrado. Este `contentDescription` parámetro se utiliza para la accesibilidad, proporcionando una descripción del ícono para usuarios con discapacidad visual. Se aplica un tono azul con este `tint` parámetro y se añade un comportamiento mediante un `Modifier` parámetro para que el ícono sea cliqueable.

### Imagen

Aquí, el `Text` elemento componible se muestra junto con una imagen usando un `painter` argumento para establecer un recurso de imagen. Es más general, ya `Icon` que puede cargar tanto imágenes de mapa de bits como PNG, JPEG y WebP, como recursos vectoriales. También puede aplicar diferentes escalas mediante el `contentScale` parámetro:

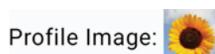


Figura 3.5 – Una imagen componible

Esta visualización se logra utilizando un `Image` componible:

```
Row(verticalAlignment = Alignment.CenterVertically) {  
    Text(  
        text = "Profile Image:",  
        Modifier.padding(end = 4.dp)
```

```

        )
    Image(
        modifier = Modifier.height(20.dp),
        painter = painterResource(id = R.drawable.sunflower)
        contentDescription = "Sunflower",
        contentScale = ContentScale.Inside
    )
}

```

La imagen es restringido dentro de los límites impuestos por la altura para restringirse dentro del espacio disponible usando `ContentScale.Inside`.

## Campos de entrada de texto

Estos elementos corresponden a la `EditText` vista en XML. Forman el núcleo de las aplicaciones para introducir texto editable. El elemento base de estos elementos componibles es `BasicTextField`. El `TextField` elemento componible es un elemento con más funciones. Elemento componible que utiliza el estilo Material Design, mientras que `OutlinedTextField`, como su nombre indica, proporciona el borde y el estilo Material Design. `BasicTextField` Es un elemento de entrada de texto minimalista que permite una personalización completa. El uso de estos elementos con el texto inicial se muestra a continuación:

```

BasicTextField(value = "Enter text:", onValueChange = {
    // Update Text
})
TextField(value = "Enter text:", onValueChange = {
    // Update Text
})
OutlinedTextField(value = "Enter text:", onValueChange = {
    // Update Text
})

```

La siguiente es una `BasicTextField` pantalla predeterminada con texto alineado al centro:

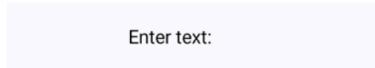


Figura 3.6 – `BasicTextField` con texto de marcador de posición

El mismo contenido se muestra aquí con `TextField`:



Figura 3.7 – Campo de texto con texto de marcador de posición

Lo mismoEl contenido se muestra aquí con `OutlinedTextField`:



Figura 3.8 – `OutlinedTextField` con texto de marcador de posición

Ya puedes verNo hay ningún estilo en `BasicTextField`. El color de fondo que se muestra es el del tema de la aplicación. `TextField` Agrega el estilo Material pre-determinado y `OutlinedTextField` un borde.

## Caja

`Checkbox` se utiliza comouna opción binaria, generalmente donde un valor pre-determinadoSe asume y permite aceptar o rechazar una opción específica. Suele aparecer en listas y formularios que solicitan el consentimiento para el uso de cookies o la aceptación de los términos y condiciones.

```

Checkbox(
    checked = false, /* or true */
    onCheckedChange = { /* action when changed */ }
)

```

Elsiguientes un ejemplo de una visualización predeterminada `Checkbox`:

Choosing to opt in to something?

Figura 3.9 – Casilla de verificación en estado desmarcado

`Switch` es similar en alcance.

## Cambiar

`Switch` es similar a `Checkbox` en que esSe utiliza para una elección binaria, pero normalmente se utiliza paraopciones basadas en funciones que se pueden activar o desactivar, afectando todo el estado o comportamiento de la aplicación:

```
Switch(  
    checked = false, /* or true */  
    onCheckedChange = { /* action when changed */ }  
)
```

El siguiente es un ejemplo de `Switch` visualización:

Feature driven choice: 

Figura 3.10 – Interruptor en estado apagado

Se pueden lograr selecciones no binarias con `Slider`.

## Control deslizante

`Slider` se utiliza paraestablecer o responder a un conjunto de valores de escala oPorcentaje de un estado en una escala lineal. Se suele usar para funciones como el brillo de una pantalla, el volumen de un dispositivo o para mostrar el progreso de una canción o película. Tiene un `steps` parámetro que permite restringir los valores a un número definido.

El siguiente es el código para mostrar un control deslizante establecido en cero con dos pasos, excluyendo el inicio y el final:

```
Slider(  
    value = 0f, /*Set to zero */  
    onValueChange = { /* action when changed */ },  
    steps = 2  
)
```

Esto produce la siguiente pantalla con la `Slider` pantalla configurada en 0:

Set or update linear value: 

Figura 3.11 – Control deslizante establecido en 0 o el valor de escala más pequeño

## Botón de radio

`RadioButton` se utiliza paraestablecer un valor único a partir de un número fijo de opciones predefinidas. Algunos ejemplos son opciones múltiples con un número determinado de respuestas, un color favorito de una lista y un método de pago al final de una compra en línea:

```
RadioButton(  
    selected = true, /* or false */  
    onClick = { /* action when clicked */ },  
)
```

La siguienteLa figura muestra tres `RadioButton` elementoscon uno seleccionado:

Favorite Color  
 Red



Figura 3.12 – Opciones de botón de opción con etiquetas

El estado seleccionado de cada botón debe manejarse manualmente para que cuando se seleccione uno, se deseleen los demás.

## Indicadores de progreso

Un comúnPatrón para informar al usuario que se está cargando una página en su lugarUna pantalla en blanco se puede usar con un indicador de progreso. Los dos indicadores de progreso integrados, personalizables por color y tamaño de trazo, son barras de progreso giratorias y lineales.

```
// Spinner Progress Bar
CircularProgressIndicator()
// Linear Progress Bar
LinearProgressIndicator(
    modifier = Modifier.fillMaxWidth()
)
```

La imagenEn la parte superior se encuentra el indicador de progreso circular. La imagen enLa parte inferior es el indicador de progreso lineal:



Figura 3.13 – Barras de progreso

A continuación, entendamos `AlertDialog`.

## AlertDialog

Los diálogos sonElementos de interfaz de usuario elevados que ocupan el foco completoDe la pantalla. Se utilizan a menudo para informar al usuario de una condición, como un mensaje de error. `AlertDialog` Normalmente muestra dos opciones: confirmar o descartar la opción descrita en el título y el mensaje:

```
AlertDialog(
    onDismissRequest = {
        /* Handle click outside the dialog window */
    },
    title = { Text("Dialog Title") },
    text = { Text("Dialog Message") },
    confirmButton = {
        Button(onClick = { /* Action on confirm */ }) {
            Text(text = "OK")
        }
    },
    dismissButton = {
        Button(onClick = { /* Action on dismiss */ }) {
            Text(text = "Cancel")
        }
    }
)
```

Este códigoresultados enLa siguiente pantalla con los botones **Cancelar** y **OK**:

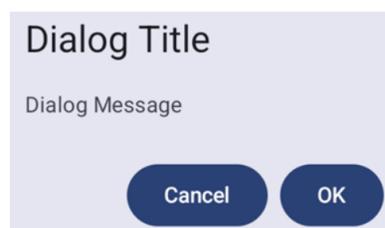


Figura 3.14 – AlertDialog con título, mensaje y botones

La `dismissButton` devolución de llamada se invoca al pulsar el botón **Cerrar** y `confirmButton` al pulsar el botón **Aceptar**. Al hacer clic fuera del cuadro de diálogo o del botón **Atrás**, este se cierra y `onDismissRequest` se invoca la devolución de llamada. Todas estas acciones impiden que el cuadro de diálogo aparezca en la pantalla.

### Ejercicio 3.03 – Creación de una pantalla de Configuración

La mayoría de las aplicaciones tiene una pantalla **de Configuración** donde se pueden configurar los ajustes de toda la aplicación. Para actualizar los ajustes en esta pantalla se requieren todos los componentes que acabamos de examinar:

1. Abre Android Studio y selecciona "Nuevo proyecto" en la pantalla de bienvenida de Android. Selecciona "Actividad vacía" y llámala "`Settings`".
2. Crea un `SettingsContainer` elemento componible con una columna para mostrar la página y una vista previa componible para que, a medida que construyes la página de forma incremental, puedas ver cómo toma forma:

```
@Composable
fun SettingsContainer(modifier: Modifier = Modifier) {
    Column(
        verticalArrangement = Arrangement.Center,
        horizontalAlignment =
            Alignment.CenterHorizontally,
        modifier = modifier
    ) {
    }
}

@Preview
@Composable
fun SettingsContainerPreview() {
    SettingsContainer()
}
```

3. Añadir `SettingsContainer` como contenido de `Scaffold`:

```
Scaffold(
    modifier = Modifier.fillMaxSize()
) { innerPadding ->
    SettingsContainer(
        modifier = Modifier.padding(innerPadding)
    )
}
```

4. Añade las siguientes cadenas al `strings.xml` archivo:

```
<string name="settings_icon_description">
    Settings
</string>
<string name="settings_consent">
    Accept non-essential cookies?
</string>
<string name="settings_mobile_data">
    Download using mobile data?
</string>
<string name="settings_text_size">Text size:</string>
<string name="settings_name">Name:</string>
<string name="sign_out">Sign out</string>
<string name="alert_title">Sign out</string>
<string name="alert_message">Are you sure?</string>
<string name="ok">OK</string>
<string name="cancel">Cancel</string>
<string name="payment_method">
    Preferred payment method
</string>
<string name="settings_profile_image">Profile Image</string>
```

5. Añade un `TextStyle` al final de `Theme.kt`:

```
val HeaderTextStyle = TextStyle(
    fontSize = 28.sp,
    fontWeight = FontWeight.ExtraBold
)
```

Hemos examinados parámetros del `Text` componente anterior y vi que `style` es uno de ellos. Usaremos el `style` parámetro para aplicarlo `HeaderTextStyle` al encabezado al crearlo.

6. Agregue un `SettingsHeader` elemento componible para mostrar el título de la página con un ícono de configuración:

```
@Composable
fun SettingsHeader() {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 14.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.Center
    ) {
        Text(
            text = stringResource(
                id = R.string.app_name),
            style = HeaderTextStyle,
            modifier = Modifier.padding(end = 10.dp)
        )
        Icon(
            imageVector = Icons.Default.Settings,
            contentDescription = stringResource(
                id = R.string.settings_icon_description
            ),
        )
    }
}
```

Cada uno de los elementos de la pantalla **Configuración** constarán de un `Row` elemento componible con dos elementos componibles dentro. En el encabezado, configuramos el `Row` elemento componible para que ocupe todo el ancho usando `fillMaxWidth()` el `Modifier` parámetro y añadimos espacio vertical arriba y abajo `Row` con `padding(vertical = 14.dp)`. El contenido del `Row` elemento componible se centra verticalmente y se organiza en el centro mediante los parámetros `verticalAlignment = Alignment.CenterVertically` y `horizontalArrangement = Arrangement.Center`. El estilo que añadimos en *el paso 5*, `HeaderTextStyle`, ahora se usa en el `Text` elemento componible para mostrar un título adecuado. Finalmente, se añade un engranaje de `configuración` `Icon` al `Row` elemento componible.

7. Añade el componible que acabas de crear a `SettingsContainer` y hazlo para todos los siguientes componibles que creemos:

```
Column(
    verticalArrangement = Arrangement.Center,
    horizontalAlignment = Alignment.CenterHorizontally,
    modifier = modifier
) {
    // Header
    SettingsHeader()
}
```

8. Agregar un componible para mostrar una imagen de perfil. Puedes descargarla desde <https://packt.link/fxKe0> o usar la tuya:

```
@Composable
fun SettingsImage() {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .padding(start = 16.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement =
            Arrangement.SpaceBetween
    ) {
        Text(
            text = stringResource(
                id = R.string.settings_profile_image),
            fontSize = 18.sp,
        )
        Image(
            modifier = Modifier.padding(
                end = 10.dp).height(34.dp)
```

```

        .clickable {
            /* Handle changing the profile image */
        },
        painter = painterResource(
            id = R.drawable.sunflower),
        contentDescription = stringResource(
            id = R.string.settings_profile_image),
    )
}
}

```

Se ha añadido el relleno adecuado para desplazar el `Row` elemento componible desde el borde inicial (izquierdo) con `padding(start = 16.dp)`, y los elementos componibles dentro del `Row` elemento componible se organizan con `Arrangement.SpaceBetween`, de modo que cualquier espacio disponible se separa entre ellos. Se ha añadido la imagen de perfil y se hizo clic mediante el ajuste `.clickable` del `Modifier` parámetro.

9. A continuación, agregue un `Checkbox` elemento componible para permitir que el usuario acepte o no las cookies no esenciales:

```

@Composable
fun SettingsCheckbox() {
    var isChecked by remember { mutableStateOf(false) }
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .padding(start = 16.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement =
            Arrangement.SpaceBetween
    ) {
        Text(
            text = stringResource(
                id = R.string.settings_consent),
            fontSize = 18.sp,
        )
        Checkbox(
            checked = isChecked,
            onCheckedChange = { isChecked = it },
        )
    }
}

```

10. Como este es el primer componible que contiene un valor sujeto a cambios. Debemos crear un estado que contenga el valor comprobado de `Checkbox`. Inicialmente, se configura `false` con lo siguiente:

```
var isChecked by remember { mutableStateOf(false) }
```

11. Cuando se cambia la casilla de verificación, `onCheckedChange` se ejecuta el bloque de función:

```
onCheckedChange = { isChecked = it },
```

Aquí `it` se muestra el valor actualizado de la casilla de verificación. Dado que el `isChecked` estado se utiliza para establecer el estado de la casilla de verificación en [nombre del elemento] `checked = isChecked`, al actualizarse, esta debe recomponerse para mostrar el estado actualizado, ya sea marcada o no.

12. Agregue un `Row` elemento componible para que el usuario cambie a la descarga a través de datos móviles, con un `Switch` elemento componible para representar los estados `sí` y `no`:

```

@Composable
fun SettingsSwitch() {
    var isChecked by remember { mutableStateOf(false) }
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .padding(start = 16.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement =
            Arrangement.SpaceBetween
    ) {
        Text(
            text = stringResource(
                id = R.string.settings_data),
            fontSize = 18.sp,
        )
        Switch(
            checked = isChecked,
            onCheckedChange = { isChecked = it },
        )
    }
}

```

```

        Text(
            text = stringResource(
                id = R.string.settings_mobile_data),
            fontSize = 18.sp,
        )
        Switch(
            modifier = Modifier.padding(end = 10.dp),
            checked = isChecked,
            onCheckedChange = { isChecked = it },
        )
    }
}

```

El comportamiento es casi el mismo que en el `Checkbox` ejemplo anterior. La única diferencia es que el elemento componible ahora se ha cambiado a `Switch` componible.

13. Añadir un `Slider` componible para configurar el tamaño del texto:

```

@Composable
fun SettingsSlider() {
    var sliderValue by remember { mutableStateOf(0f) }
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp)
            .padding(start = 16.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement =
            Arrangement.SpaceBetween
    ) {
        Text(
            modifier = Modifier.padding(end = 16.dp),
            text = stringResource(
                id = R.string.settings_text_size),
            fontSize = 18.sp,
        )
        Slider(
            value = sliderValue,
            onValueChange = { sliderValue = it },
            steps = 2
        )
    }
}

```

De nuevo, la manera de actualización `Slider` es muy similar a `Checkbox` y `Switch`. En este caso, sin embargo, el estado del control deslizante está disponible `sliderValue` como float (`of`), que es un número de punto flotante (que puede contener un decimal) y se actualiza en `onValueChange {}`, lo que indica que el control deslizante es una escala y no tiene un estado binario de *activado* o *desactivado*. En este ejemplo, lo hemos especificado con dos pasos, lo que, al añadirse a los ajustes de inicio y fin, significa que hay cuatro tamaños de texto disponibles para que el usuario elija.

14. Añade `RadioButton` componibles para seleccionar el método de pago preferido:

```

@Composable fun SettingsRadioButtons() {
    var selectedPaymentMethod by remember { mutableStateOf("") }
    Column(modifier = Modifier
        .fillMaxWidth()
        .padding(16.dp)) {
        Text(text = stringResource(id = R.string.payment))
        modifier = Modifier.padding(bottom = 8.dp))
        listOf("PayPal", "Credit Card", "Bank Transfer").  

            paymentMethod -> Row(
                verticalAlignment = Alignment.CenterVertically,
                modifier = Modifier.padding(vertical = 4.dp))
        ) {
            RadioButton(
                selected = (selectedPaymentMethod ==
                    paymentMethod),
                onClick = { selectedPaymentMethod =
                    paymentMethod },
                colors = RadioButtonDefaults.colors())
        }
        Text(text = paymentMethod, modifier =
            Modifier.padding(start = 8.dp))
    }
}

```

```
        }
    }
}
```

Se crean tres elementos en unLista con `listOf`. Luego, se recorren para crear un `RadioButton` elemento componible para cada método de pago. PayPal se establece con el valor predeterminado con `var selectedPaymentMethod by remember { mutableStateOf("PayPal") }`, y luego `RadioButton` se evalúan los elementos componibles para comprobar si están seleccionados con `selected = (selectedPaymentMethod == paymentMethod)`. Luego, `onClick` se añade para seleccionarlos si se hace clic en ellos con `selectedPaymentMethod = paymentMethod`.

15. Por último, añade un `AlertDialog` archivo componible para cerrar la sesión del usuario con una confirmación para verificar que es su intención cerrar la sesión:

```
@Composable fun SettingsAlertDialog() {
    var showDialog by remember { mutableStateOf(false) }
    Button(onClick = { showDialog = true }) {
        Text(text = stringResource(id = R.string.sign_out))
    }
    if (showDialog) {
        AlertDialog(
            onDismissRequest = { showDialog = false },
            title = { Text(text = stringResource(id =
                R.string.alert_title)) },
            text = { Text(text = stringResource(id =
                R.string.alert_message)) },
            confirmButton = {
                Button(onClick = { showDialog = false })
                    Text(text = stringResource(id = R.str
                )
            },
            dismissButton = {
                Button(onClick = { showDialog = false })
                    Text(text = stringResource(id =
                        R.string.cancel))
            }
        )
    }
}
```

La decisión de mostrar o no el diálogo se rige por `showDialog`, que es establecido en `false` por `mutableStateOf(false)`. Cuando else hace clic en el botón, `showDialog` se cambia a `true` y `AlertDialog` se muestra con un título y un mensaje. `confirmButton` y `dismissButton` permite realizar acciones cuando se presiona cualquiera de ellos y también cierra el `AlertDialog` elemento componible.

Cuando haya agregado los componentes anteriores, la pantalla debería aparecer como en la siguiente figura:

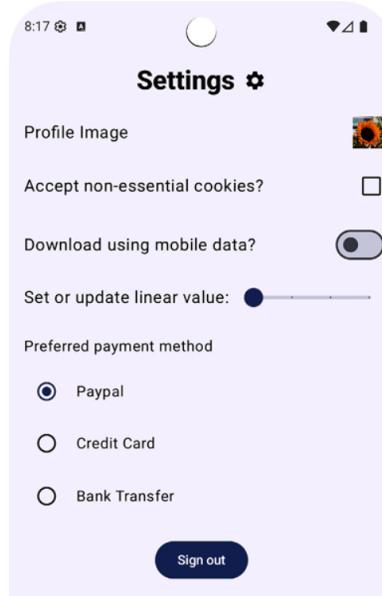




Figura 3.15 – Pantalla de configuración

Este ejercicio ha demostrado cómo es fácil mostrar elementos de la interfaz de usuario de Compose, aplicarles estilo y añadir comportamiento. Junto con los campos de entrada de texto componibles, estos son los componentes principales de una interfaz de usuario de Jetpack Compose. Son los nodos hoja donde se captura el comportamiento y desde donde se propagan las acciones.

A continuación, exploraremos cómo estos bloques de construcción componibles se pueden posicionar en grupos de diseño componibles.

## Grupos de diseño de Jetpack Compose

Ya has utilizado los grupos de diseño de elementos componibles clave del `Row` primer `Column` capítulo. Con estos elementos componibles, puedes crear muchos diseños complejos. Sin embargo, existen otros grupos de diseño que son fundamentales para crear interfaces de usuario atractivas. Primero, abordaremos el `Box` diseño componible, que permite posicionar elementos en una pila con la alineación de sus elementos componibles secundarios. A continuación, analizaremos el `Surface` grupo de diseño componible, un contenedor basado en Material Design que permite desarrollar el tema de la aplicación para personalizar y mejorar el aspecto de la interfaz de usuario añadiendo fondo y elevación. A continuación, analizaremos el `Card` grupo de diseño componible, una superficie especializada personalizada con Material Design para diseños tipo tarjeta. Después, profundizaremos en los `Column` elementos `Row` componibles.

### Caja

El `Box` grupo de diseño coloca los componibles dentro de él usando una pila que es alineado con los bordes del elemento principal. Por lo tanto, el último elemento componible añadido se muestra arriba.

Al agregar tres `Text` componibles de "RED", "GREEN", y "BLUE" contenidos dentro de un `Box` componible, se muestran uno encima del otro:

```
Box(Modifier.fillMaxSize()){  
    Text(text = "RED",  
        fontSize = 20.sp,  
    )  
    Text(text = "GREEN",  
        fontSize = 20.sp,  
    )  
    Text(text = "BLUE",  
        fontSize = 20.sp,  
    )  
}
```

Lo siguiente es un `Box` componible que muestra cada `Text` elemento componible colocado uno encima del otro:

BREEN

Figura 3.16 – Cuadro que muestra elementos componibles apilados

💡 Consejo rápido : ¿Necesitas ver una versión de alta resolución de esta imagen? Abre este libro en el Packt Reader de nueva generación o consúltalo en formato PDF/ePub.

💡 El Packt Reader de última generación y una copia gratuita de este libro en formato PDF/ePub están incluidos con tu compra. Escanea el código QR o visita [packtpub.com/unlock](http://packtpub.com/unlock) y usa la barra de búsqueda para encontrar este libro por nombre. Comprueba la edi-



De forma predeterminada, una `Box` posición componiblesus elementos en la esquina superior izquierda (`contentAlignment:Alignment = Alignment.TopStart`), pero esto se puede anular configurando el `contentAlignment` parámetro en el `Box` elemento componible en sí:

```
Box(contentAlignment = Alignment.Center)
```

Sin embargo, el contenido seguiría apilado. La alternativa es configurar la alineación de los elementos componibles dentro del `Box` elemento componible:

```
@Composable fun BoxDisplay() {
    Box(Modifier.fillMaxSize(), contentAlignment = Alignment.
        Text(
            text = "RED",
            fontSize = 20.sp,
            fontWeight = FontWeight.Bold,
            modifier = Modifier.align(Alignment.TopStart)
        )
        Text(
            text = "GREEN",
            fontSize = 20.sp,
            fontWeight = FontWeight.Bold,
        )
        Text(
            text = "BLUE",
            fontSize = 20.sp,
            fontWeight = FontWeight.Bold,
            modifier = Modifier.align(Alignment.BottomEnd)
        )
    }
}
```

Este código produce la siguiente pantalla:



Figura 3.17 – Cuadro que muestra diferentes valores de alineación

En este ejemplo, se utilizan modificadores para definir el `Box` propio elemento componible y su `Box` contenido. Esto permite definir el comportamiento y la disposición del elemento como componible o componibles dentro de él.

apariencia del elemento, grupo componible y componentes dentro de él.

Los cuadros son muy útiles cuando necesitas un posicionamiento simple de elementos componibles que usan una de las nueve posiciones de alineación de `TopStart`, `TopCenter`, `TopEnd`, `CenterStart`, `Center`, `CenterEnd`, `BottomStart`, `BottomCenter`, y `BottomEnd`.

El próximo grupo de diseño componible para explorar es `Surface`.

## Superficie

`Surface` es un diseño básico componible que le permite proporcionar una base para el contenido de una aplicación. Puedes agregar elementos de estilo como fondo, forma, elevación y bordes, envolviendo un único elemento componible incrustado:

```
@Composable fun SurfaceExample(){
    Surface(
        modifier = Modifier
            .padding(40.dp)
            .width(300.dp),
        color = Color.LightGray,
        contentColor = Color.Yellow,
        shape = RoundedCornerShape(22.dp),
        tonalElevation = 8.dp,
        shadowElevation = 8.dp,
        border = BorderStroke(2.dp, SolidColor(Color.Blue))
    ) {
        Text(
            text = "Hello, Surface!",
            textAlign = TextAlign.Center,
            modifier = Modifier
                .padding(16.dp).fillMaxWidth(),
            fontSize = 22.sp
        )
    }
}
```

El elemento componible anterior `Surface` tiene un color de fondo de `color.Gray` y el estilo predeterminado de su contenido será el `contentColor` color de `Color.Yellow`. La forma se establece en `RoundedCornerShape` con un tamaño de esquina de `22.dp`, y hay otras formas disponibles, como `Circle`. Se puede añadir una temática de material específica usando `tonalElevation` y `shadowElevation`. Hay una gama de opciones de borde que permiten usar degradados lineales y radiales. Aquí, se aplica un color sólido:



Hello, Surface!

Figura 3.18 – Grupo de diseño de superficie con contenido de texto

Cercanamente relacionado a `Surface` es lo `Card` componible.

## Tarjeta

El `Card` componible es un relleno `Surface` que tiene un estilo predefinido para el relleno, la elevación y los bordes. El siguiente código demuestra cómo crear un componible de aspecto muy similar, pero que establece menos parámetros:

```
Card(
    modifier = Modifier.padding(40.dp)
        .width(300.dp),
    colors = CardColors(
        Color.LightGray,
        Color.Yellow,
        Color.LightGray,
        Color.Yellow
    ),
    border = BorderStroke(2.dp, SolidColor(Color.Blue))
) {
    Text(
        text = "Hello, Card!",
        textAlign = TextAlign.Center,
        modifier = Modifier.padding(16.dp).fillMaxWidth(),
        fontSize = 22.sp
    )
}
```

`tonalElevation` y `shadowElevation` se establecen con valores predeterminados, y el `colors` parámetro se establece en `cardColors`, que establece los valores `color` y, como `contentColor` así como los colores correspondientes para cuando el botón esté deshabilitado.



Figura 3.19 – Grupo de diseño de tarjeta con contenido de texto

Se puede lograr un diseño más estructurado para múltiples elementos utilizando `Column` y `Row` componibles.

## Columna

Tú tienesUsé este diseño componible desde el primer capítulo. Su función es distribuir los componibles en una columna, controlando el inicio y el fin. Alineación del contenido con el `horizontalAlignment` parámetro y su disposición vertical con el `verticalArrangement` parámetro. Las `horizontalAlignment` opciones son `Start`, `End`, o `CenterHorizontally` para posicionar los elementos componibles en el eje x. La disposición del eje y permite colocar el contenido de diferentes maneras. El contenido se puede espaciar con una distancia fija mediante el `spacedBy` parámetro, o si se utiliza `Modifier.fillMaxSize()` `Modifier.fillMaxHeight()`, se puede espaciar con los siguientes valores: `Top`, `Bottom`, `Center`, `SpaceEvenly`, `SpaceAround` o `SpaceBetween`.

La siguienteEl código utiliza tres `Text` elementos componibles para ilustrar cómo se distribuye el contenido con un `verticalArrangement` parámetro, que es `SpaceEvenly`:

```
Column(  
    modifier = Modifier.fillMaxSize(),  
    horizontalAlignment = Alignment.CenterHorizontally,  
    verticalArrangement = Arrangement.SpaceEvenly  
) {  
    Text(  
        text = "RED",  
        color = Color.White,  
        fontSize = 24.sp,  
        modifier = Modifier  
            .background(Color.Red)  
            .width(100.dp)  
            .padding(4.dp),  
        textAlign = TextAlign.Center  
)  
    Text(  
        text = "GREEN",  
        color = Color.White,  
        fontSize = 24.sp,  
        modifier = Modifier  
            .background(Color.Green)  
            .width(100.dp)  
            .padding(4.dp),  
        textAlign = TextAlign.Center  
)  
    Text(  
        text = "BLUE",  
        color = Color.White,  
        fontSize = 24.sp,  
        modifier = Modifier  
            .background(Color.Blue)  
            .width(100.dp)  
            .padding(4.dp),  
        textAlign = TextAlign.Center  
)  
}
```

La siguienteLa figura muestra cómo `SpaceEvenly` se distribuye el espacio.uniformemente entre los contenidos, y también antes del primer y después del último elemento:



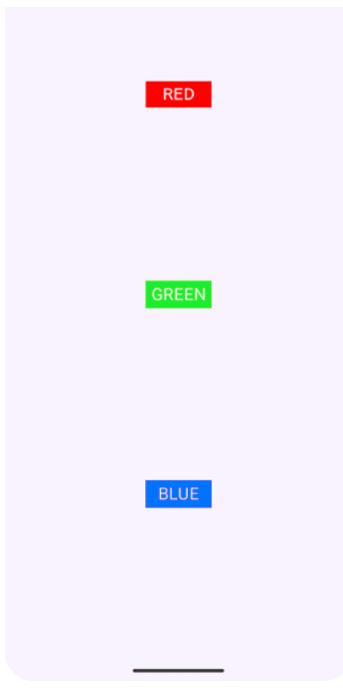


Figura 3.20 – Contenido de las columnas espaciado uniformemente

EsteLa ilustración de la documentación oficial muestra cómo se pueden usar todas las opciones pantalla: <https://packt.link/6RJve>.

El diseño que lo acompaña `Column` es componible a `Row`.

## Fila

La organización de los elementos componibles a lo largo del eje `x` es lo que `Row` hace. Utiliza parámetros para la organización y alineación del contenido de forma similar a `Column`. El `verticalAlignment` parámetro establece la alineación del contenido en `Top`, `CenterVertically`, o `Bottom`, y `horizontalArrangement` define cómo se distribuirá el contenido del área que ocupa la fila de principio a fin, si `Modifier.fillMaxSize()` o `Modifier.fillMaxWidth` está definido. Estas opciones son `Start`, `End`, `Center`, `SpaceEvenly`, `SpaceAround` y `SpaceBetween`.

También puedes usar `spacedBy` para agregar espacio despuésCada elemento de la fila. El siguiente código muestra cómo `Row` se compone el contenido `SpaceAround`, de modo que se añade espacio alrededor de cada elemento, y el espacio antes y después del último elemento es la mitad del espacio entre los elementos secundarios:

```
Row(
    modifier = Modifier.fillMaxSize(),
    horizontalArrangement = Arrangement.SpaceAround,
    verticalAlignment = Alignment.CenterVertically
) {
    Text(
        text = "RED",
        color = Color.White,
        fontSize = 24.sp,
        modifier = Modifier
            .background(Color.Red)
            .width(100.dp)
            .padding(4.dp),
        textAlign = TextAlign.Center
    )
    Text(
        text = "GREEN",
        color = Color.White,
        fontSize = 24.sp,
        modifier = Modifier
            .background(Color.Green)
            .width(100.dp)
            .padding(4.dp),
        textAlign = TextAlign.Center
    )
    Text(
        text = "BLUE",
        color = Color.White,
        fontSize = 24.sp,
        modifier = Modifier
            .background(Color.Blue)
            .width(100.dp)
            .padding(4.dp),
        textAlign = TextAlign.Center
    )
}
```

```

        text = "BLUE",
        color = Color.White,
        fontSize = 24.sp,
        modifier = Modifier
            .background(Color.Blue)
            .width(100.dp)
            .padding(4.dp),
        textAlign = TextAlign.Center
    )
}

```

Este Row componible se verá así:

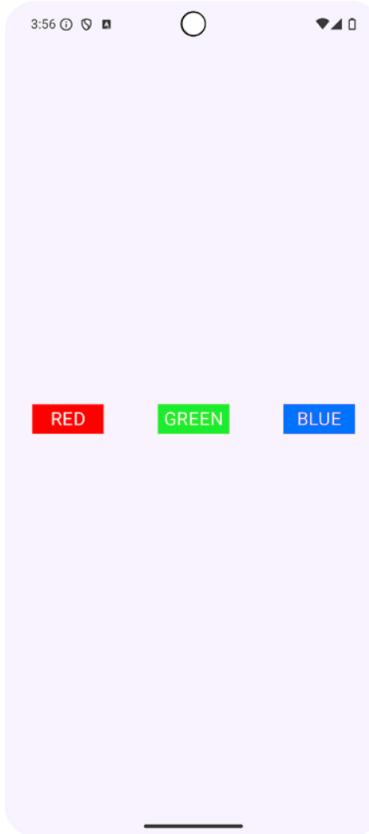


Figura 3.21 – Contenido de las filas espaciadas alrededor

Esta ilustración de la documentación oficial muestra cómo se muestran todas las opciones: <https://packt.link/jYNX6>.

Los grupos principales de diseño componibles proporciona los componentes básicos que usarás para crear la interfaz de usuario. Para verlos en acción, sigue el siguiente ejercicio para crear una página de perfil.

#### Ejercicio 3.04 – Creación de una página de perfil

Aplicaciones que dependen de la interacción entre usuarios y suele tener una página de perfil. Crearás una de estas páginas de perfil en Jetpack Compose:

1. Abre Android Studio y selecciona "Nuevo proyecto" en la pantalla de bienvenida de Android. Selecciona "Actividad vacía" y llámalo "Profile".
2. Crea un elemento componible llamado `Profile` con una vista previa:

```

@Composable
fun Profile(modifier: Modifier) {
}
@Preview(showBackground = true)
@Composable
fun ProfilePreview() {
    Profile(modifier = Modifier.padding(20.dp))
}

```

Añade un `Profile` elemento componible y la vista previa para que, al empezar a crear la pantalla, puedas ver cómo se muestra. Por defecto, Android Studio muestra un fondo transparente en las vistas previas. Como crearás un `Surface` para

una `Surface` transparente en las vistas previas. Como creas un `Surface` elemento componible para mostrar la tarjeta, para distinguirla como tal, necesitas elevación. Esto produce una sombra, que es difícil de ver en un fondo transparente.

3. Agregue una `Surface` elevación componible con esquinas redondeadas para crear el panel básico en el que se mostrará el contenido de la tarjeta:

```
Surface(  
    modifier = modifier  
        .fillMaxWidth()  
        .padding(16.dp),  
    shape = RoundedCornerShape(16.dp),  
    shadowElevation = 8.dp,  
    color = Color.White  
) {}
```

Como hayNo se ha añadido ningún contenido `Surface` en esta etapa; la vista previa solo mostrará el fondo. Una vez añadido el contenido, Verá `RoundedCornerShape` el efecto. Esto se especifica como `16.dp`, que establece el mismo tamaño para todas las esquinas. `shadowElevation` Establece la elevación de la superficie sobre el contenido principal, lo cual se logra aplicando sombra. El color de fondo se especifica como `Color.White`.

4. Como el contenido se presenta de arriba a abajo, necesitamos crear un `Column` elemento componible. Queremos que haya relleno interior y que todo el contenido esté alineado horizontalmente por defecto:

```
Column(  
    horizontalAlignment = Alignment.CenterHorizontally,  
    modifier = Modifier  
        .padding(16.dp)  
        .fillMaxWidth()  
) {}
```

Como tienes algo de contenido, aunque sea un grupo de diseño vacío, puedes ver la elevación:

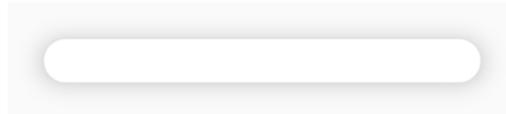


Figura 3.22 – Columna vacía componible que se muestra dentro de una superficie componible

5. El contenido comenzará mostrando una imagen de perfil y una forma de editarla. Queremos que el botón **Editar** sea superpuesto ligeramente sobre la parte superior de la imagen, por lo que crea un `Box` elemento componible con un tamaño definido:

```
Box(modifier = Modifier.size(116.dp)) {
```

6. Agregar unaImagen de perfil dentro del cuadro:

```
Image(  
    // Replace with your image resource  
    painter = painterResource(id = R.drawable.cat),  
    contentDescription = "Profile Picture",  
    contentScale = ContentScale.Inside,  
    modifier = Modifier  
        .padding(16.dp)  
        .size(100.dp)  
        .clip(RoundedCornerShape(50.dp))  
)
```

Puedes agregar tu propia imagen o descargar una del repositorio:  
<https://packt.link/efhEs>

Para mayor accesibilidad, agregue un `contentDescription` parámetro y luego el `Modifier` parámetro con `clip(RoundedCornerShape(50.dp))` para recortar la imagen con una forma de esquinas redondeadas con un tamaño de `50.dp`. El `ContentScale` valor se especifica como `ContentScale.Inside`, por lo que la

valor se especifica como `componenamiento`, por lo que la imagen se escalará. Si es necesario, se puede reducir el tamaño en el `Image` elemento componible. Se puede hacer clic en la imagen para actualizar la imagen de perfil. El orden del relleno es importante. Actualmente, `Image` tieneEspacio aplicado entre su borde y el contenido interior. Así, la imagen del gato puede ocupar todo el `100.dp` tamaño.

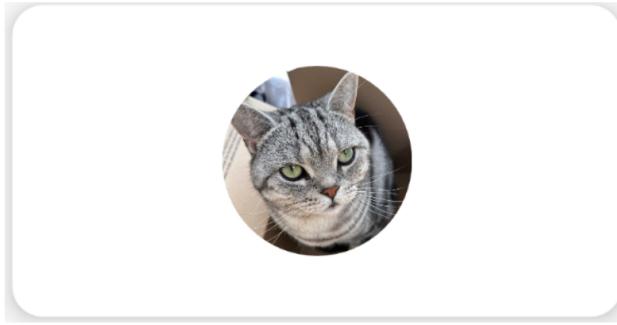


Figura 3.23 – Relleno de imagen aplicado antes de especificar la altura

7. Ahora, intercambie estos valores:

```
.size(100.dp)  
.padding(16.dp)
```

Solo existe el `size` valor, `100.dp`, menos el `padding` valor de `16.dp`, por lo que `84.dp` en total se muestra la imagen.

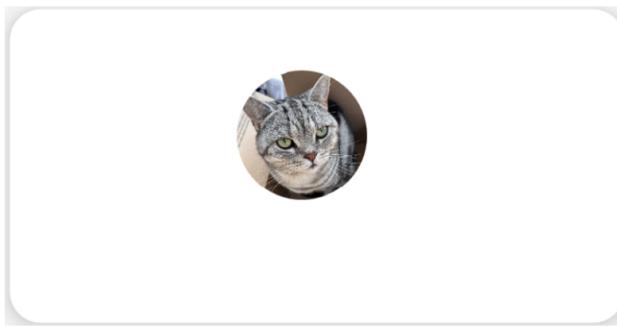


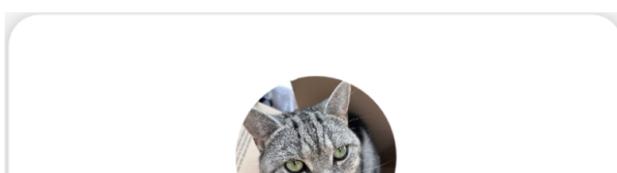
Figura 3.24 – Relleno de imagen aplicado después de especificar el tamaño

De esta manera la imagen se muestra más pequeña.

8. Intercambiar el valores de nuevo y Agregue un `card` elemento componible en el que se pueda hacer clic con un `Text` elemento componible debajo del `Image` elemento componible:

```
Card(  
    onClick = {/**/},  
    modifier = Modifier.align(BottomEnd),  
    border = BorderStroke(  
        1.dp, SolidColor(Color.Blue))  
) {  
    Text(  
        text = "Edit",  
        modifier = Modifier.padding(horizontal = 4.dp),  
        fontSize = 12.sp)  
}
```

Como el `Card` componible se alinea usando la `BottomEnd` opción del `Box` componible que lo envuelve, el `Text` componible aparece en la parte inferior de la imagen alineado al final:



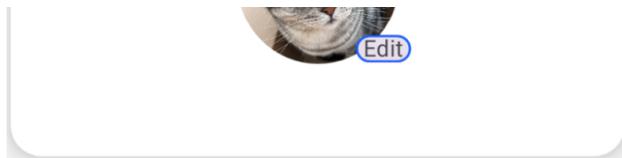


Figura 3.25 – Relleno de imagen aplicado después de especificar la altura

A continuación, vamos a Añade algunos detalles personales para el perfil.

9. Agregue dos `Text` elementos componibles para el nombre y la descripción (debajo del `Box` elemento componible), uno encima del otro y luego un `Row` elemento componible.con dosBotones con **llamadas a la acción** ( `CTAs` ) para seguir y enviar mensajes al usuario:

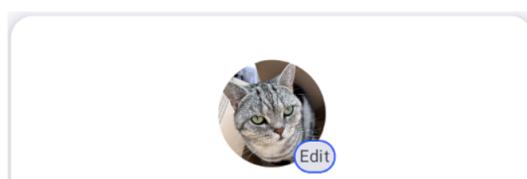
```
Spacer(modifier = Modifier.height(16.dp))
Text(
    text = "Jane Doe",
    fontSize = 24.sp,
    fontWeight = FontWeight.Bold,
    textAlign = TextAlign.Center,
    modifier = Modifier.padding(top = 16.dp)
)
Text(
    text = "Mobile Developer | Tech Enthusiast",
    fontSize = 16.sp,
    color = Color.Gray,
    modifier = Modifier.padding(vertical = 16.dp)
)
Row(
    horizontalArrangement = Arrangement.SpaceEvenly,
    modifier = Modifier.fillMaxWidth()
) {
    Button(onClick = { /* TODO: Add follow action */ }) {
        Text(text = "Follow")
    }
    Button(onClick = { /* TODO: Add message action */ }) {
        Text(text = "Message")
    }
}
```

EresUsar un `Spacer` elemento componible para añadir espacio vertical entre los elementos de la columna. Esto suele ser muy útil para personalizar la visualización, ya que no es necesario añadir relleno a otros elementos para lograr el aspecto deseado. `HorizontalDivider` Y `VerticalDivider` lograr efectos similares con una línea divisoria. Se especifica individualmente el `Text` aspecto mediante parámetros en lugar de estilos.espaciando horizontalmente los componibles con un `horizontalArrangement` valor de `SpaceEvenly`, de modo que cualquier espacio disponible se posicionará de manera equitativa con los botones a lo largo del eje horizontal.

10. El últimoEl paso es agregar el `Profile` componible `setContent` dentro del `Theme` componible y el `Scaffold` componible:

```
setContent {
    ProfileTheme {
        Scaffold(
            modifier = Modifier.fillMaxSize()
        ) { innerPadding ->
            Profile(
                modifier = Modifier
                    .padding(innerPadding)
            )
        }
    }
}
```

La exhibición finalDebería verse así:



Jane Doe  
Mobile Developer | Tech Enthusiast  
Follow Message

Figura 3.26 – Página de perfil

Este ejercicioSe ha demostrado el uso de la mayoría de los grupos de diseño principales de elementos componibles. Sin embargo, hay más grupos de diseño por descubrir. En [el Capítulo 4](#), "Creación de la Navegación de la App", usarás [nombre Scaffold del grupo] para diseñar el contenido de la app en secciones más manejables y también usarás grupos de diseño de navegación para navegar por tus apps. Aprenderás a crear listas y cuadrículas en el [Capítulo 6](#), "Creación de Listas con Jetpack Compose". Para un posicionamiento más preciso de los elementos componibles, consulta la versión de Jetpack Compose del diseño heredado basado en vistas XML [nombre del grupo] ConstraintLayout. Proporciona un control posicional más preciso de los elementos de la IU entre sí, así como respecto a las directrices y barreras: <https://packt.link/u8tol>.

Ahora que has adquirido elConocimiento de los grupos de diseño y elementos componibles principales, complete la actividad para crear un tablero comercial que muestre métricas financieras clave.

### Actividad 3.01 – Creación de un panel de métricas empresariales

El objetivo de este ejercicio es utilizar algunos de los componentes principales para mostrar cuatro negocios clave.Métricas en una pantalla de panel. Debe tener cuatro mosaicos que muestren una métrica principal y un área de visualización detallada debajo de los mosaicos que...muestra más detalles sobre la métrica seleccionada:

1. Abre Android Studio y selecciona "Nuevo proyecto" en la pantalla de bienvenida de Android. Selecciona "Actividad vacía" y llámala " Dashboard".
2. Añade un Text componible con Business Dashboard como encabezado de la página.
3. Cree cuatro estados para las cuatro métricas principales. Estos pueden ser cardenas, enteros u otro tipo de dato.
4. Crea un MutableState objeto con la remember función que contendrá el valor de la métrica seleccionada.
5. Cree un mosaico de panel único componible que se reutilizará para mostrar una métrica principal de la empresa. Debe incluir un texto de título y un controlador de clic para la acción al hacer clic en el mosaico.
6. Crea un elemento componible que tome la métrica seleccionada y evalúe su valor antes de mostrar una de las métricas clave en detalle. Puede contener varias líneas de texto.
7. Cree el cuerpo principal del tablero, que mostrará los cuatro mosaicos del tablero con un título de métrica clave y un controlador de clic que actualiza la métrica seleccionada a uno de los cuatro estados.
8. Verifique que cada uno de los cuatro mosaicos complete los detalles de la empresa para cada uno de los cuatro estados correspondientes.
9. Para una tarea adicional, cambie el color de fondo del título del titular seleccionado para mostrar;Qué métrica de la empresa?se está mostrando.



La solución de esta actividad se puede encontrar en  
<https://packt.link/PAeqs>.

### Resumen

Este capítulo ha abordado en profundidad los componentes básicos y los grupos de diseño. Comenzamos con una comparación entre la creación de una interfaz de usuario (UI) con XML tradicional y los componentes. Posteriormente, estudiamos los componentes básicos, antes de finalizar con un estudio de los principales grupos de diseño. Estos son los componentes fundamentales para la creación de interfaces de usuario (UI) de Android. Puedes aprovechar estos conceptos para avanzar y crear interfaces de usuario cada vez más avanzadas.

interfaz de usuario establecidos para crear una navegación clara y consistente en sus aplicaciones.

Desbloquea los beneficios exclusivos de este libro ahora

[UNLOCK NOW](#)

Escanee este código QR o vaya a [packtpub.com/unlock](http://packtpub.com/unlock), luego busque este libro por nombre.

*Nota : Tenga a mano la factura de compra antes de comenzar.*



Capítulo anterior

< [Creación de flujos de pantalla...](#)

Siguiente capítulo

[Creación de la navegación de... >](#)



Búsqueda colapsada