

JETPACK COMPOSE 2025

BEGOÑA RODRÍGUEZ FERRERAS
brodfer@gmail.com



Jetpack Compose

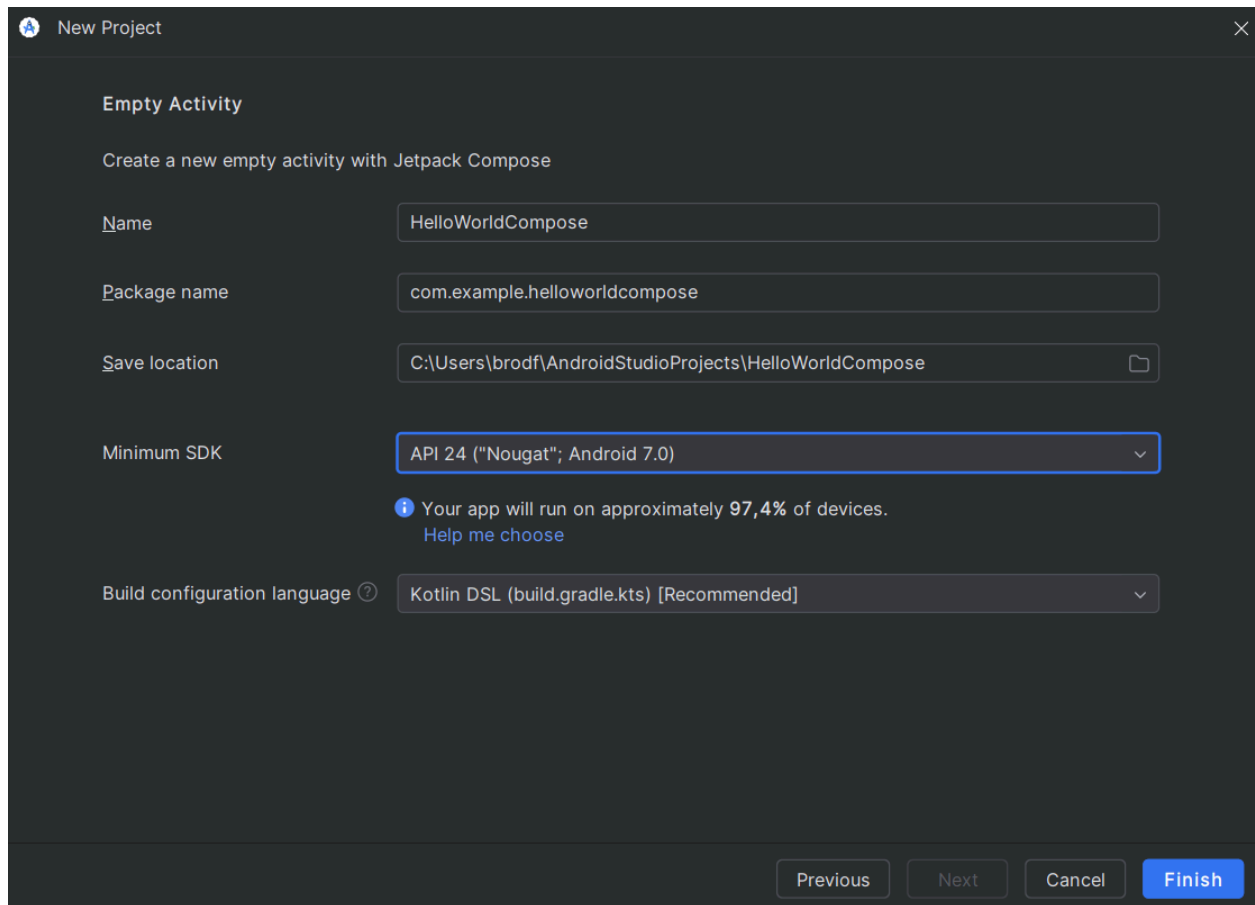
Jetpack Compose es un kit de herramientas moderno para crear IU nativas de Android. Menos código que además será más sencillo de entender.

Los componentes de IU se compilan con **funciones declarativas**. No se edita ningún diseño XML ni se usa el editor de diseño. En cambio se llaman a funciones que admiten composición para definir qué elementos se quieren y el compilador de Compose hará el resto.

Primer proyecto

La **versión mínima de Android Studio será Arctic Fox** y para crear el proyecto tendremos que elegir **Empty Activity**. Puede cambiar el nombre según la versión de Android así que buscad que tenga el símbolo de Jetpack Compose en el interior.

La API Mínima será la API 21(No debería dar problemas ya esto) y si os dais cuenta ya no da la opción de elegir el lenguaje entre Java y Kotlin. Será **Kotlin**.

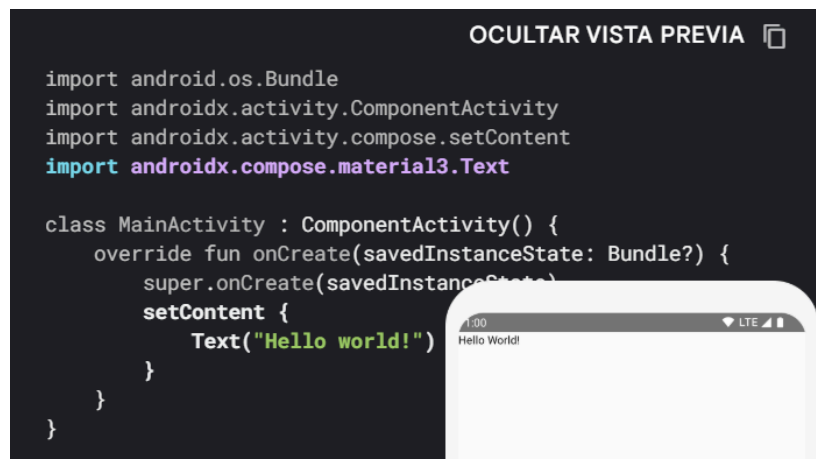


El MainActivity existe al igual que en todos los proyectos que hayáis creado previamente, pero ahora ya no vais a tener un activity_main.xml y tendréis un setContent. **Dentro de setContent solo podremos llamar a funciones que sean Composable.**

La plantilla predeterminada ya contiene algunos elementos de Compose, vamos a tratar de verlos uno a uno.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldComposeTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

Hay elementos adicionales que las actividades incluyen a día de hoy, pero esto sería totalmente funcional como el HolaMundo más simple, llamamos a la función `Text()` y le pasamos el texto que queremos escribir:



The screenshot shows a code editor with the following Kotlin code for `MainActivity`:

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material3.Text

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello world!")
        }
    }
}
```

At the top right of the editor, there is a button labeled "OCULTAR VISTA PREVIA" with a square icon. To the right of the code, a preview of the app interface is shown, displaying "Hello World!" on a white background with a status bar at the top.

La función de componibilidad `Text` ya está definida por la biblioteca de la IU de Compose y muestra una etiqueta de texto en la pantalla.

HelloWorldComposeTheme

En primer lugar tendréis un tema definido por defecto (*HelloWorldComposeTheme*). Jetpack Compose tiene por defecto una API de Temas para poder definir el estilo de

la aplicación: [Material Design 3](#). Permite personalizar **colores, formas y tipografías** para que se usen de manera consistente en la aplicación y haya compatibilidad con varios **temas**, como el **claro** y el **oscuro**. En este caso sería solo la declaración del tema, no le estamos especificando ningún parámetro todavía, de manera que recibiremos el estilo predeterminado de “modelo de referencia”. [Más info sobre los temas](#).

Scaffold

En Material Design, un [andamiaje](#) es una estructura fundamental que proporciona una plataforma estandarizada para interfaces de usuario complejas. Une diferentes partes de la IU, como las barras de la app y los botones de acción flotantes, lo que les da a las apps un aspecto y estilo coherentes.

Para pasar contenido a un Scaffold se pasa un valor `innerPadding` a la `lambda` `content` que puedes usar en elementos secundarios componibles.

Nota: Según la versión de Android con la que estemos trabajando nos puede aparecer por defecto `Surface` o `Scaffold` dado que es algo que se cambió a partir de la introducción de Material Design 3 en Abril de 2024.

Al final la diferencia es que un Scaffold puede incluir una barra de herramientas en la parte superior, un botón de acción flotante y una barra de navegación en la parte inferior.

@Composable (Funciones de componibilidad)

Las funciones de componibilidad son los componentes básicos de una IU en Compose. Una función de componibilidad cumple con lo siguiente:

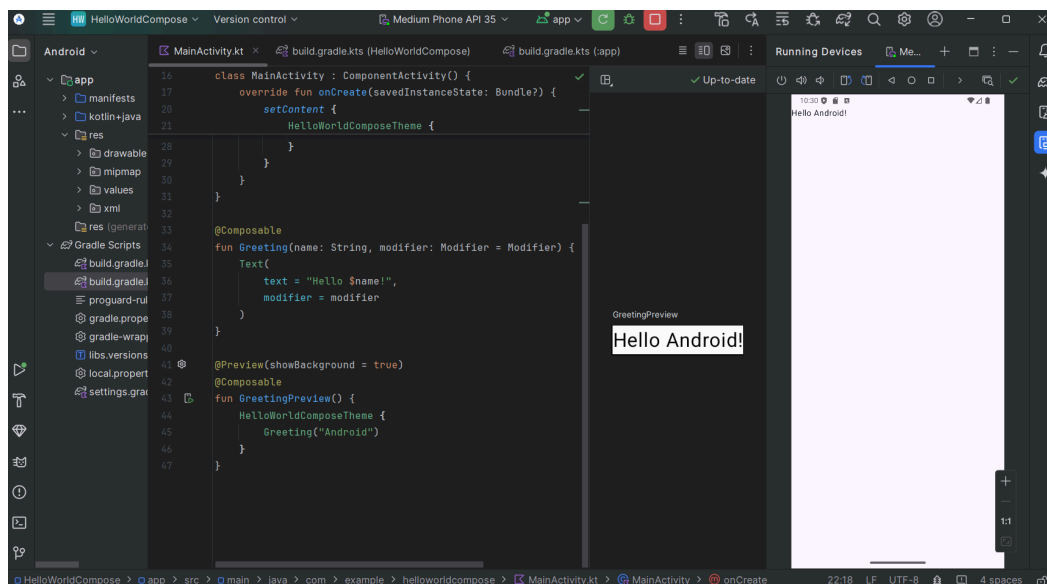
- Describe alguna parte de tu IU.
- No devuelve nada.
- Toma información y genera lo que se muestra en la pantalla.

Para definir una función como componable tendremos que añadir esta etiqueta. Nuestra función tiene dos argumentos uno el texto name que es el que le pasamos arriba y otro que es el Modifier que veremos más tarde en detalle.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

@Preview (Vistas previas)

La anotación preview nos permite ver los cambios sin necesidad de compilar e instalar la app en un emulador o dispositivo Android. Por ejemplo si en vez del texto "Android" le pasamos DAM veréis que los cambios se reflejan en la vista sin necesidad de compilar de nuevo.



Hay algunas limitaciones como el hecho de que no podemos pasar parámetros directamente a una función con la etiqueta preview, por eso se llama a través de la función `GreetingPreview()` que **no recibe ningún parámetro**.

Importante: **Esta anotación se debe usar en una función de componibilidad y que no acepte parámetros.**

Otro detalle importante es que la propia etiqueta Preview puede admitir parámetros, como el nombre que se mostrará en la pantalla con el argumento `name` o que se muestre más parecido al dispositivo: `showSystemUi = true`.

```
@Preview(
    showBackground = true,
    showSystemUi = true,
    name = "My Preview")
@Composable
fun GreetingPreview() {
    HappyBirthdayTheme {
        Greeting( name: "Android")
    }
}
```

Debido a la forma en que se renderizan las vistas previas en Android Studio, son livianas y no requieren todo el framework de Android para renderizarlas. Sin embargo, esto tiene las siguientes limitaciones:

- No tienes acceso a la red.
- No tienes acceso al archivo.
- Es posible que algunas APIs de Context no estén completamente disponibles.

Podéis encontrar mucha más información sobre las preview aquí:

<https://developer.android.com/develop/ui/compose/tooling/previews?hl=es-419#:~:>

[text=If%20you%20need%20to%20display%20the%20status%20and,to%20change%20the%20behavior%20of%20the%20preview%20accordingly.](#)

Modificadores

Es como un atributo extra que pueden recibir nuestras funciones, para meter **todos los atributos que normalmente se meterían en un XML** (el padding, la longitud, la elevación...).

Vamos a crear un nuevo @Composable para que lo veáis desde cero.

Nota: la unidad de .dp va a ser necesario importarla la primera vez que se use.

```
@Preview(showBackground = true)
@Composable
fun ejemploModifier(){
    Text(text="Modificador", modifier = Modifier.padding(24.dp))
}
```

Si ponemos por ejemplo solo en horizontal nos aparecerá padding en horizontal



Lo bueno de trabajar con Compose es que no hay problemas de modificaciones. Porque cuando cambiemos el estado de un componente, por ejemplo el texto, la vista se va a regenerar, así que no nos va a dar problemas de falsa información.

Varios temas importantes sobre los modificadores:

1. Se pueden encadenar. Será común encontrar cadenas de este estilo:

```

@Composable
private fun Greeting(name: String) {
    Column(
        modifier = Modifier
            .padding(24.dp)
            .fillMaxWidth()
    ) {
        Text(text = "Hello,")
        Text(text = name)
    }
}

```

Los modifier de `.fillMaxSize()`, `.fillMaxWidth()` y `fillMaxHeight()` serán importantes. Permiten rellenar todo el espacio en las dos dimensiones o a lo largo o lo ancho.

2. El orden de los modificadores es importante. Como cada función realiza cambios en el Modifier que muestra la función anterior, la secuencia afecta al resultado final. En la documentación viene un buen ejemplo al respecto con una imagen clicable a la que se le añade padding después de clicable y permite hacer clic en el padding después:

<https://developer.android.com/develop/ui/compose/modifiers?hl=es-419>

3. La práctica recomendada es que **todos los elementos componibles acepten un parámetro modifier y pasen ese modificador al primer elemento secundario que emita la IU**. Si lo haces, tu código será más reutilizable y su comportamiento será más intuitivo y predecible.

```

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier .padding(16.dp).background(Color.LightGray).padding(8.dp)
    )
}

```

Por ejemplo, Greeting es un composable que acepta un parámetro modifier con un valor predeterminado de Modifier. Dentro del cuerpo de la función, el modifier se pasa al elemento Text, que es el primer (y único) elemento secundario.

Otro detalle, si usáis mucho un modificador lo podéis guardar en una variable.

Diseños

Cambiar el tamaño del texto

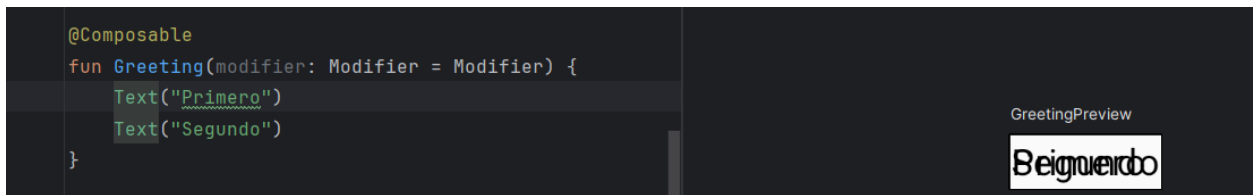
Ponemos el tamaño del texto en una fuente grande, por ejemplo 90.sp (en sp para que coja el tamaño de fuente por defecto del móvil). El texto se nos pisará por lo que tenemos que marcar una altura de línea también.



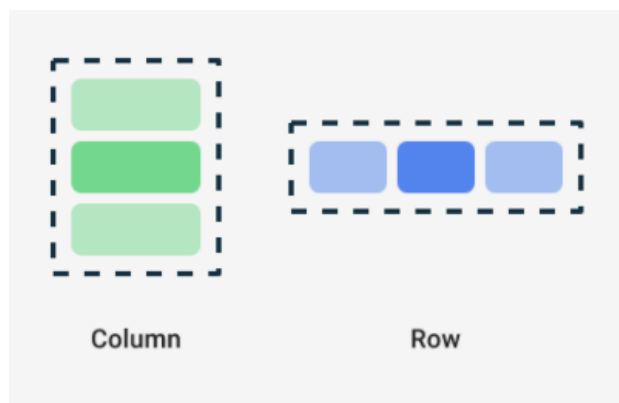
Jerarquías

La jerarquía de la IU se basa en la contención, es decir, un componente puede contener uno o más componentes. A veces, se usan los términos superior y secundario. El contexto aquí es que los elementos superiores de la IU contienen elementos secundarios de la IU, los cuales, a su vez, pueden contener elementos secundarios de la IU. En Compose, compilas una jerarquía de la IU llamando a las funciones que admiten composición desde otras funciones del mismo tipo.

Para que entendáis esto os muestro lo que ocurriría si ponemos más etiquetas de texto:



La maquetación será a través de Columnas, Filas y Cajas. Aprenderemos entonces sobre los elementos componibles Column, Row y Box, que pueden actuar como elementos superiores de la IU. Esto lo podréis imaginar como un Linear Layout.



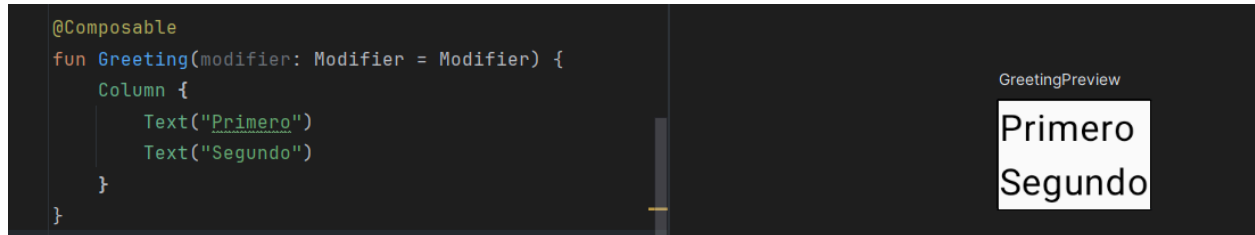
Columnas

Este sería un ejemplo de un diseño en columnas:

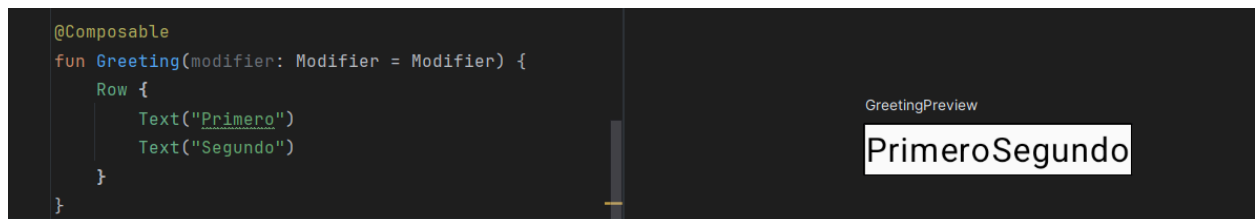


La función **Column** nos va a permitir organizar los elementos de forma vertical.

Si queremos organizar los datos en una columna, podemos hacer uso de **Column**:



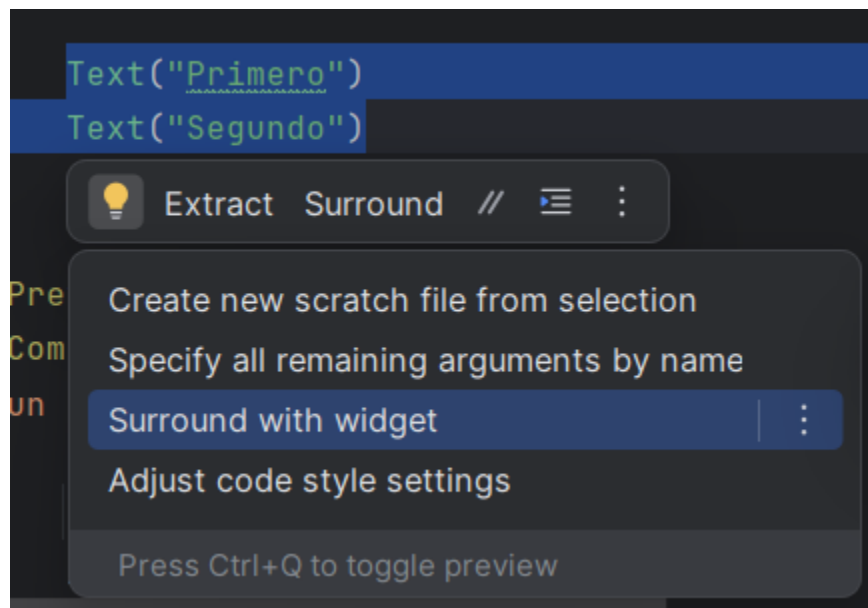
Vamos a poder usar **Row** también para organizar los elementos de manera horizontal:



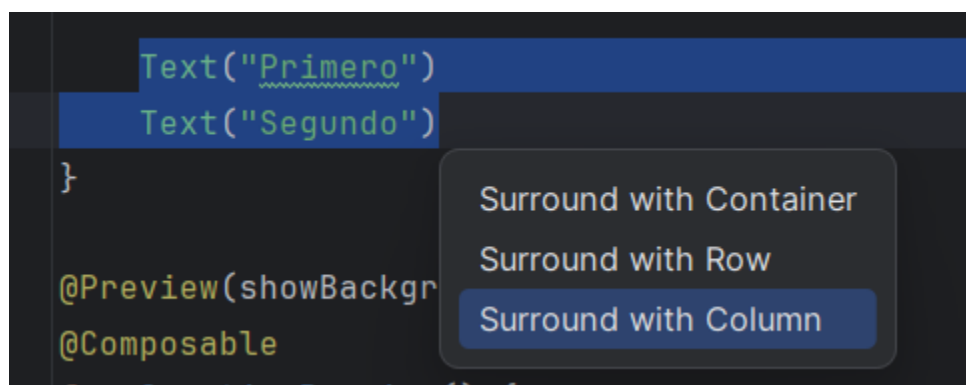
y **Box** para apilarlos. Box lo veremos un poco más adelante porque tenemos que hacer uso de modificadores y será algo más complejo.

Nota: En los fragmentos de código anterior, observad que se usan llaves en lugar de paréntesis en la funciones de componibilidad Row/Column. Esto se llama *sintaxis de expresión lambda final*. **Esta práctica es recomendada y común en Compose**, por lo que debes familiarizarte con la apariencia del código.

Una opción en Android para crear componentes de este estilo es seleccionarlos y al seleccionarlos nos aparecerá un menú contextual con una bombilla y la opción Surround with widget, haciendo clic en esta opción



Haciendo clic en esta opción nos aparecerá otra opción para rodear con filas o columnas:



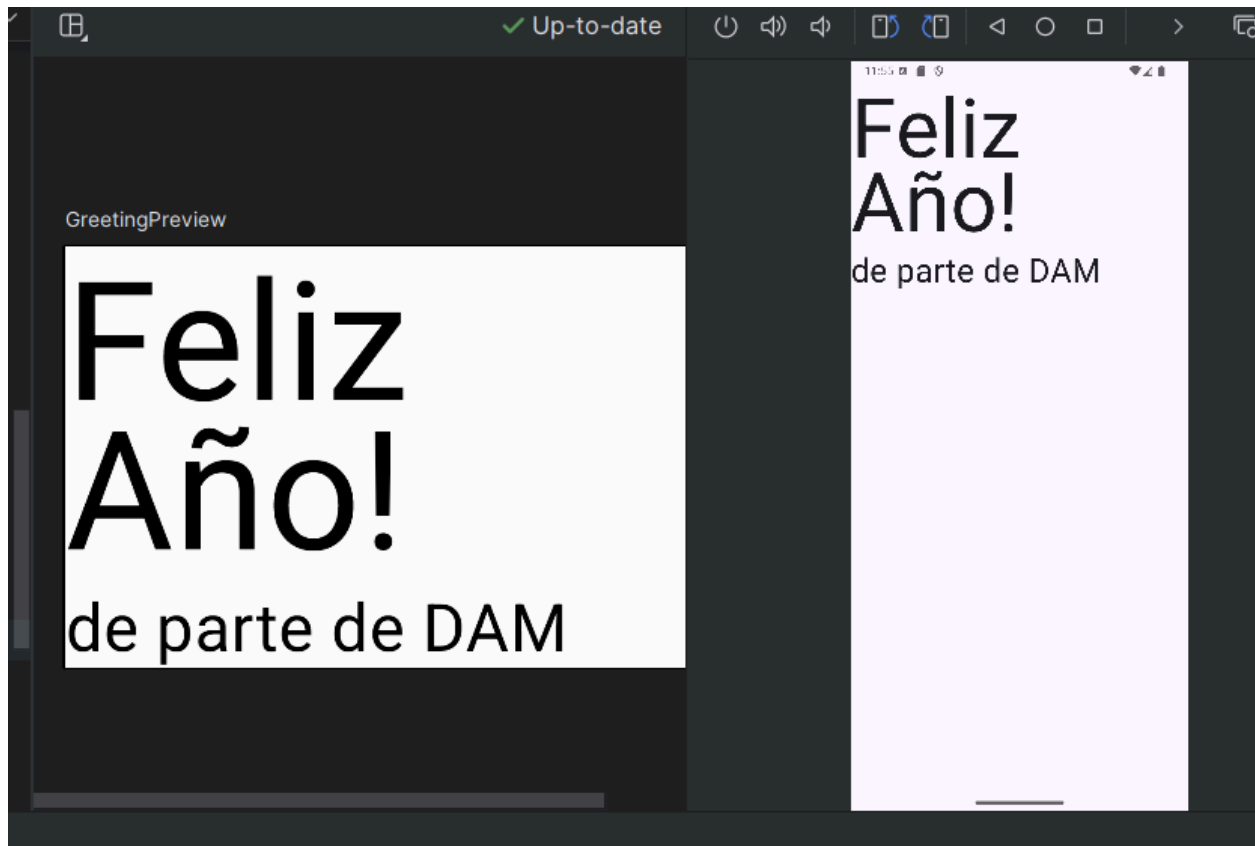
La opción de Container es la de Box.

Recordad que debéis pasar el parámetro modificador al elemento secundario en los elementos componibles. Eso significa que debes pasar el parámetro modificador al elemento Column componible.

```
@Composable
fun Greeting(modifier: Modifier = Modifier) {

    Column(modifier = modifier) {
        Text("Primero")
        Text("Segundo")
    }
}
```

Ejercicio: Quiero que con todo lo aprendido hasta ahora realices una app de cero con un contenido similar al de la pantalla:



Como centrar los textos

Para alinear el saludo en el centro de la pantalla, agrega un parámetro llamado `verticalArrangement` a la columna y configúralo en `Arrangement.Center`.

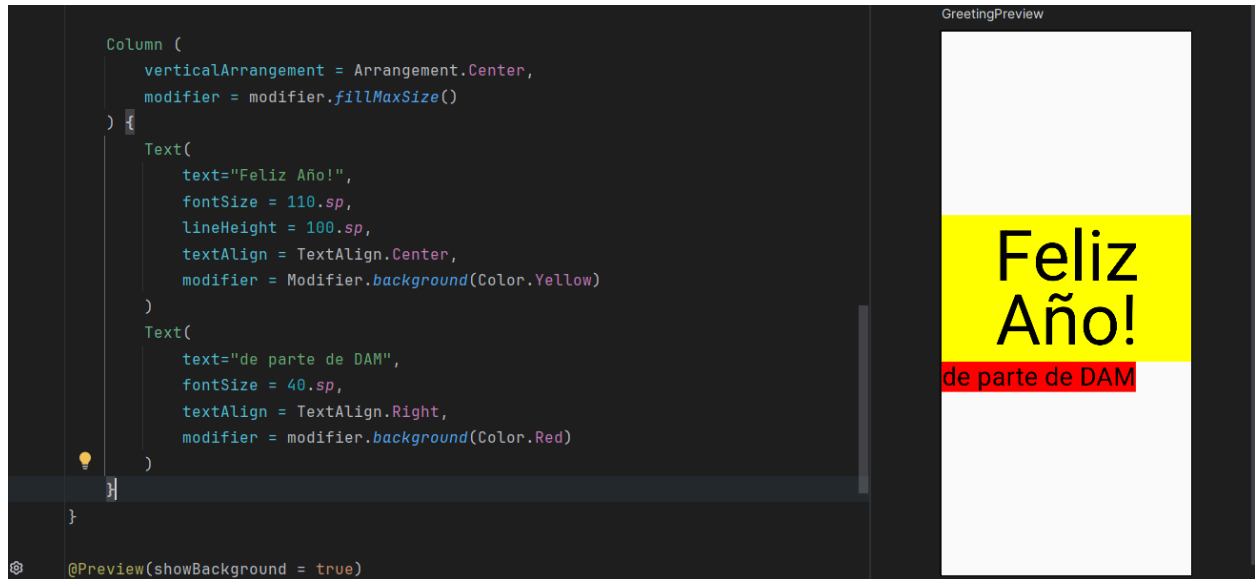
Nota: **Si no ponemos que la columna llene toda la pantalla no va a funcionar.**
Podemos poner un background a los dos textos para ver que espacio de la pantalla ocupan



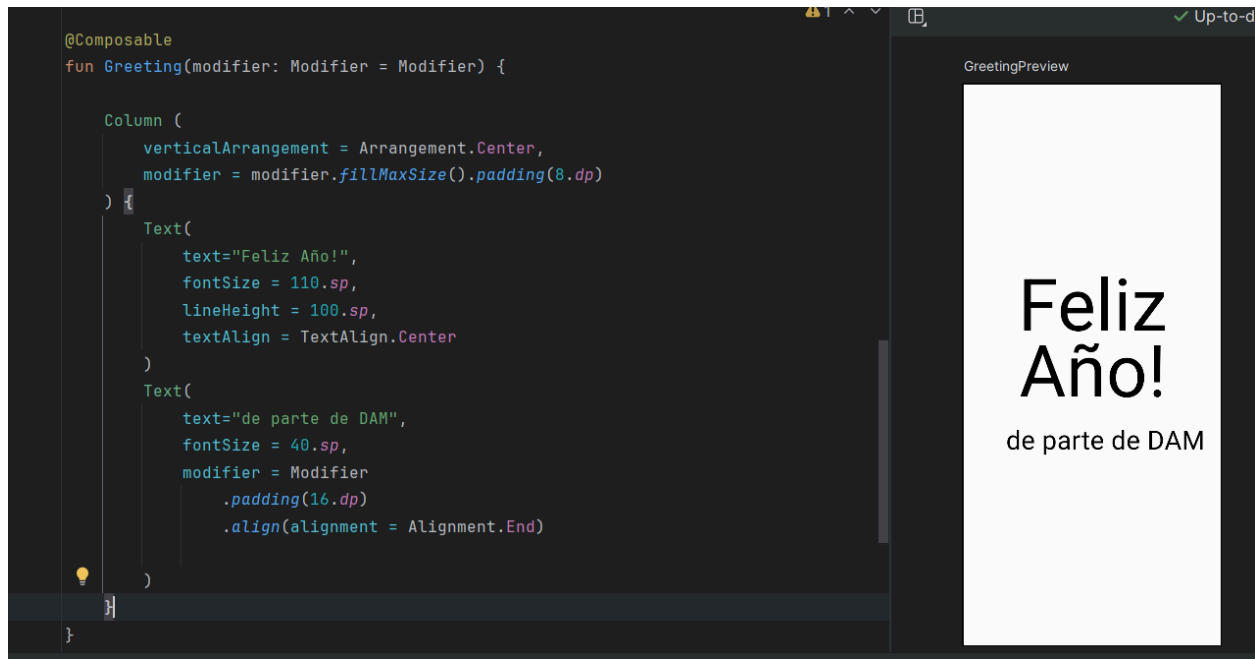
Podéis probar los distintos tipos de Arrangement habrá tanto en Vertical como en Horizontal como en común a ambos.

Para cambiar la alineación del texto Feliz Año como centrada podéis pasarle el parámetro `textAlign`.

Esto lo podemos hacer directamente porque ocupa toda la pantalla.



Pero para el texto de abajo esto no va a funcionar con la parte en rojo, porque no ocupa toda la pantalla. Habrá dos opciones: hacer con el modificador que ocupe todo el espacio y pasar el parámetro de alineación o directamente poner la alineación con un modificador.

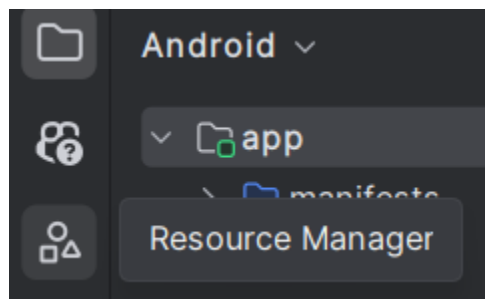


Imágenes

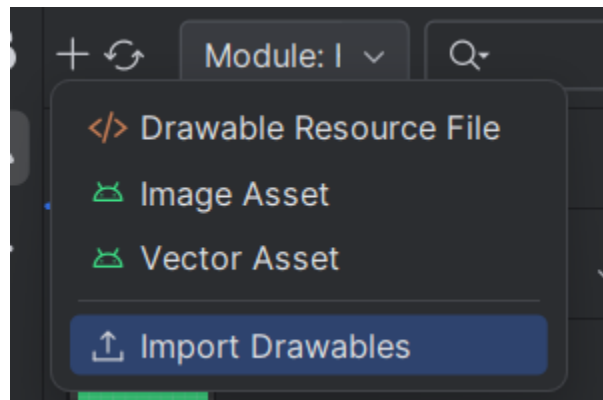
En primer lugar vamos a descargar la imagen con la que vamos a trabajar:

<https://github.com/google-developer-training/basic-android-kotlin-compose-birthday-card-app/blob/main/app/src/main/res/drawable-nodpi/androidparty.png>

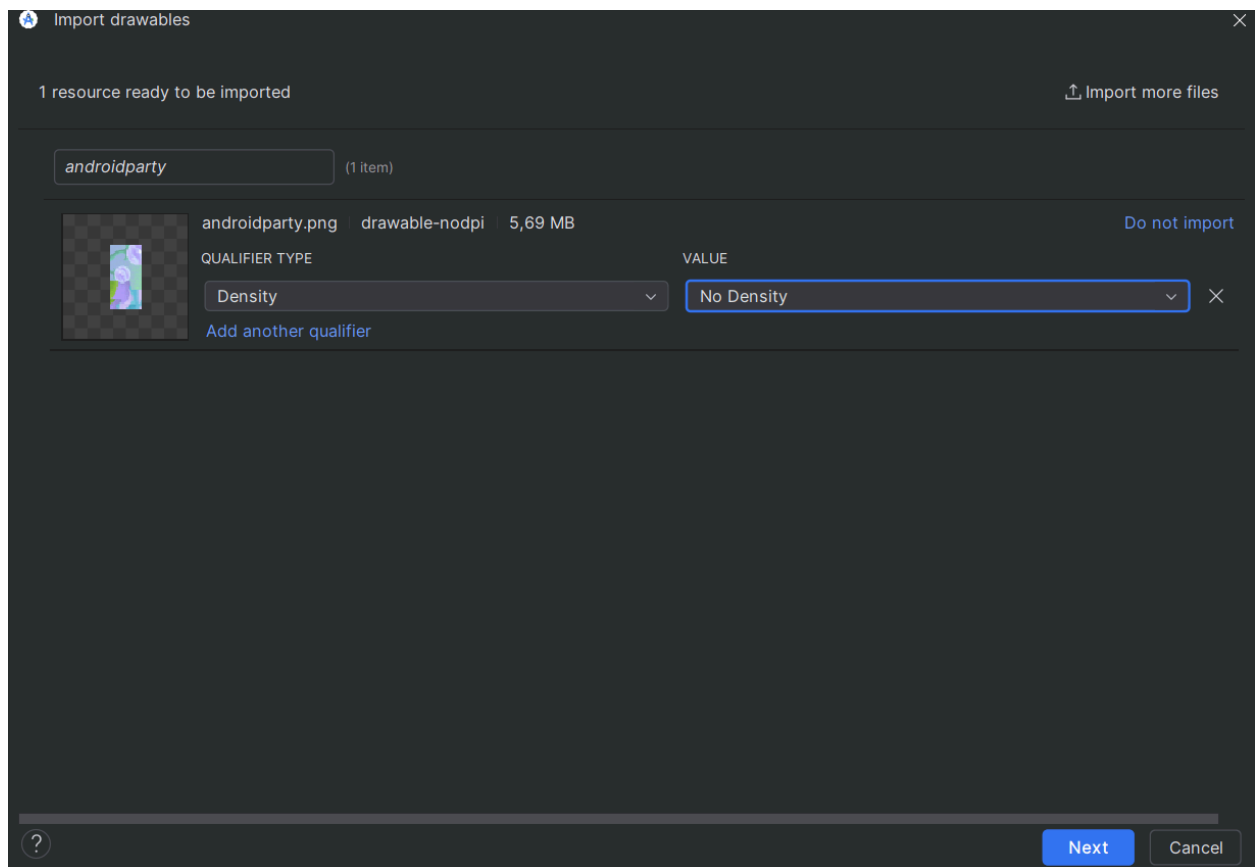
En Android Studio vamos al Resource Manager:



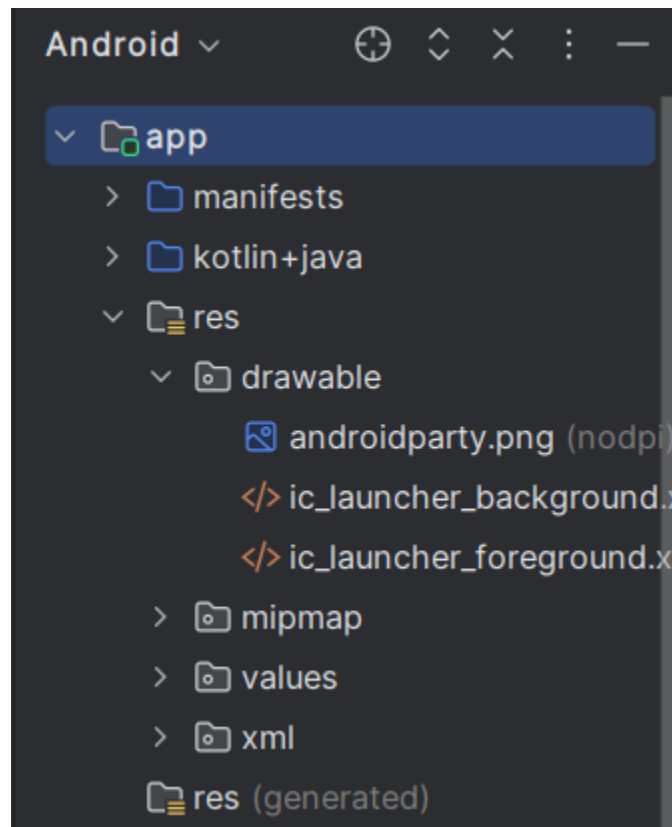
Hacemos clic en el + y en Import Drawables.



Seleccionad Density como el tipo, no nos vamos a detener en explicar los tipos, simplemente es para que se vea bien como fondo en diferentes dispositivos:

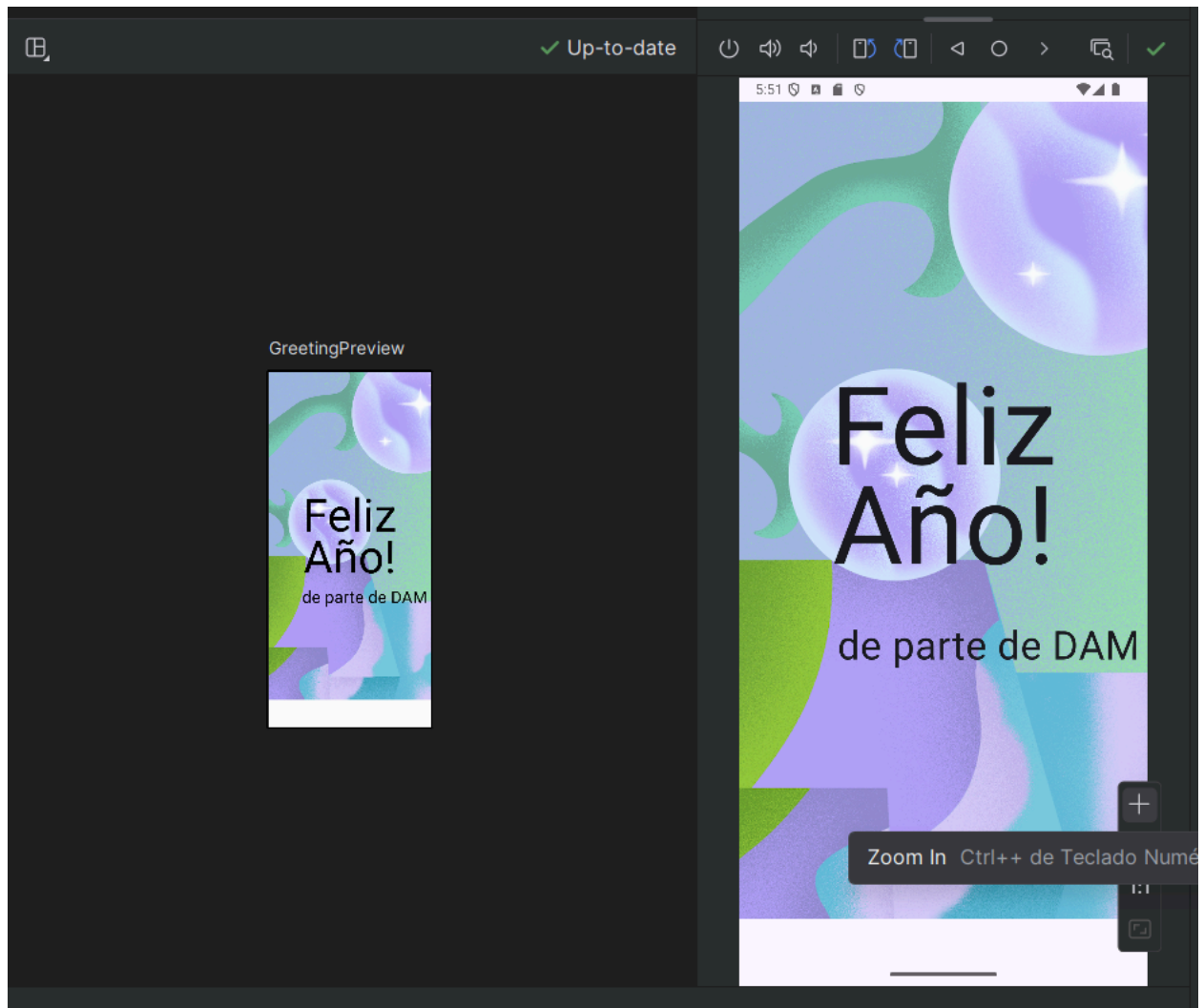


Con eso ya tendríamos la imagen disponible en nuestro proyecto para trabajar con ella:



Para insertarla en nuestro código. Primero creamos una variable en la que pasamos el recurso a la función `painterResource()`, luego pasamos esto como argumento a una función componible `Image`. El argumento `contentDescription` es por temas de accesibilidad. La imagen de esta app solo se incluye con fines decorativos por tanto se puede poner `null` u omitir.

```
val image = painterResource(R.drawable.androidparty)
Image(painter = image, contentDescription = null)
```



Lo propio cuando tenemos texto y una imagen es meterlo dentro de una **Box**, que nos va a servir **para apilar elementos uno sobre otro**.



Recordad, bombilla, surround with widget y Container en este caso:

```
@Composable
fun Greeting(modifier: Modifier = Modifier) {
    val image = painterResource(R.drawable.androidparty)
    ⚡ Box (modifier){

        Image(painter = image,contentDescription = null)

        Column(
            verticalArrangement = Arrangement.Center,
            modifier = modifier.fillMaxSize()
        ) {
            Text(
                text = "Feliz Año!",
                fontSize = 110.sp,
                lineHeight = 100.sp,
                textAlign = TextAlign.Center,

            )
            Text(
                text = "de parte de DAM",
                fontSize = 40.sp,
                textAlign = TextAlign.Right,
```

No nos vamos a detener en ver todos los argumentos de las imágenes. Pero si que quiero que seáis conscientes de que hay unos cuantos:

<https://developer.android.com/develop/ui/compose/graphics/images/customize?hl=es-419#content-scale>

Nosotros queremos ajustar la imagen a la pantalla entera así que usaremos **ContentScale.Crop**, que ajusta la escala de la imagen manteniendo la relación de aspecto y hace que se ajuste a la pantalla. Bajamos la opacidad con el parámetro alpha.

```
Image( painter = image,|
      contentDescription = null,
      contentScale = ContentScale.Crop,
      alpha = 0.5F)
```







Iconos

Importar Iconos es incluso más sencillo, dado que Material Design provee una biblioteca. Los nombres los podéis ver en: <https://fonts.google.com/icons>

```
Icon(Icons.Rounded.Menu, contentDescription = "Localized description")
```

Se pueden modificar tratándolos como objetos:

<https://developer.android.com/reference/kotlin/androidx/compose/material/icons/package-summary>

	APIs	Description	Preview
Icons	<code>Icons</code>	Icons	
Default	<code>Icons.Default</code>	Default icons	
Filled	<code>Icons.Filled</code>	Filled icons	
Outlined	<code>Icons.Outlined</code>	Outlined icons	
Rounded	<code>Icons.Rounded</code>	Rounded icons	
Two tone	<code>Icons.TwoTone</code>	Two tone icons	
Sharp	<code>Icons.Sharp</code>	Sharp icons	

Ejercicio:

Intentad hacer este diseño podéis coger los colores que queráis, poned vuestro nombre y datos. El icono lo podéis encontrar [aquí](#).

